

# GENASIS: *GENERAL* ASTROPHYSICAL *SIMULATION* SYSTEM.

## I. FUNDAMENTALS

Christian Y. Cardall<sup>1,2</sup>, Reuben D. Budiardja<sup>1,2,3,4</sup>, Eirik Endeve<sup>5</sup>, and Anthony Mezzacappa<sup>1,2,5</sup>

cardallcy@ornl.gov

### ABSTRACT

GenASiS (*General Astrophysical Simulation System*) is a new code being developed initially and primarily, though by no means exclusively, for the simulation of core-collapse supernovae on the world's leading capability supercomputers. Using the features of Fortran 2003 that allow for object-oriented programming, its classes are grouped into three major divisions: (1) *Basics*, which contains some basic utilitarian functionality for large-scale simulations on distributed-memory supercomputers; (2) *Mathematics*, which includes generic mathematical constructs and solvers that are as agnostic as possible with regard to the specifics of any particular system; and (3) *Physics*, which sets up physical spaces associated with various theories of spacetime (including gravity), defines various forms of stress-energy, and combines these into ‘universes.’ To provide a foundation for subsequent papers focusing on the implementation of various pieces of physics needed for the simulation of core-collapse supernovae and other astrophysical systems, this paper—the first in a series—focuses on *Basics* and one of the major constructs under *Mathematics*: cell-by-cell refinable *Manifolds*. Two sample problems illustrate our object-oriented approach and exercise the capabilities of the *Basics* and *Manifolds* divisions of GenASiS.

*Subject headings:* methods: numerical

### 1. INTRODUCTION

Many problems in astrophysics and cosmology will press against the boundaries of supercomputer software and hardware development for some time to come. Among the most challenging are time-dependent systems that should be treated in three position space dimensions, plus up to three momentum space dimensions for those that involve radiation transport. Several types of physics—and their couplings—must be addressed. Taken together, physics such as self-gravity, turbulent cascades, reactions that may or may not be in equilibrium, and the operation of both microscopic and macroscopic processes often imply the simultaneous relevance of multiple scales in space and time, typically requiring some kind of spatial adaptivity and multiple solvers (at least some of which may be time-implicit). Solvers

---

<sup>1</sup>Physics Division, Oak Ridge National Laboratory, Oak Ridge, TN 37831-6354, USA

<sup>2</sup>Department of Physics and Astronomy, University of Tennessee, Knoxville, TN 37996-1200, USA

<sup>3</sup>Joint Institute for Heavy Ion Research, Oak Ridge National Laboratory, Oak Ridge, TN 37831-6374, USA

<sup>4</sup>National Institute for Computational Sciences, University of Tennessee, Knoxville, TN 37996, USA

<sup>5</sup>Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN 37831-6354, USA

deployed with good resolution in as much of phase space as possible fill the memory and churn through the cycles of the largest supercomputers available. That the relevant theory is described by time-dependent partial (and sometimes integro-)differential equations implies the need for synchronous evolution and communication between different regions of position space (and sometimes momentum space). These latter aspects seem ever more difficult to address due to the fact that increases in computing capability seem only to come with such additional burdens as distributed memory and distributed (and, more recently, heterogeneous) processing capacity. From the perspective of working astrophysicists and cosmologists, it can seem that the physics itself recedes ever further away as their efforts are channeled towards developing and tailoring codes to the specific features of these increasingly complex high-performance machines. In this environment, the availability of well-designed codes with broadly applicable physics capabilities is increasingly valuable to researchers.

One such computationally demanding problem is the elucidation of the explosion mechanism of core-collapse supernovae (Mezzacappa 2005; Woosley and Janka 2005; Kotake et al. 2006; Janka et al. 2007; Janka 2012). A star with mass greater than  $\sim 8 M_{\odot}$  develops a degenerate core during its final burning stage. Once the core becomes sufficiently large—roughly the Chandrasekhar mass—it becomes unstable and undergoes catastrophic collapse on a nearly free-fall time scale. Collapse of an inner subsonic portion of the core halts around nuclear density (where nucleons begin to overlap), leading to the development of a shock wave at the interface between this inner core and the supersonically infalling outer portion of the core. The shock eventually will disrupt the entire star and give rise to the luminous supernova, but it stagnates shortly after its formation due to the endothermic reduction of heavy nuclei into their constituent nucleons and enervating neutrino losses. The mechanism of shock revival—that is, the explosion mechanism—remains to be fully elucidated by numerical simulations, but is expected to involve some combination of heating by intense neutrino fluxes streaming from the nascent neutron star, fluid instabilities, rotation, and magnetic fields. In fact, the relative contributions of these phenomena may vary from event to event: pre-supernova stars differ in properties such as mass and rotation, leading to explosions that may or may not be jet-like with an associated gamma ray burst, and either a black hole or a neutron star as a compact remnant.

Even this brief description conveys an initial sense of the multiphysics and multiscale nature of core-collapse supernova explosion mechanism simulations. The problem manifestly requires self-gravity, which must be general relativistic to treat the more rare and extreme case of black hole formation, and ideally would be at least approximately relativistic even when the compact remnant is a neutron star. The treatment of hydrodynamics—and ideally magnetohydrodynamics, especially in connection with black hole formation—must be able to handle shocks. At high density the equation of state must describe neutron-rich nuclear matter at finite temperature, and at low density it is desirable to track nuclear composition with a reaction network spanning a wide range of nuclear species. Neutrino transport must span diffusive, decoupling, and free-streaming regimes, and include several species and their interactions with each other and with various fluid constituents. Gravitational collapse, a steepening density cliff at the surface of the nascent neutron star, and regions of turbulence strongly recommend some form of spatial adaptivity. The stiff equations of neutrino transport and a nuclear network normally require time-implicit evolution. At the present time computational limitations still require dimensional reduction of phase space; simple estimates suggest that at least exascale resources will be required for the full transport problem. There is fairly wide agreement that retention of at least energy dependence in full neutrino transport is important, and that all three position space dimensions should be included, but finished simulations of this type have yet to be published by any group (though some are in progress; see for instance Bruenn et al. 2009).

GenASiS (*General Astrophysical Simulation System*) is a new code being developed, at least initially and primarily, for the simulation of core-collapse supernovae on the world’s leading capability supercomputers. ‘General’ denotes the capacity of the code to include and refer to multiple algorithms, solvers, and physics and numerics choices with the same abstracted names and/or interfaces. In GenASiS this is accomplished with features of Fortran 2003

that support the object-oriented programming paradigm (e.g. Reid 2007; Adams et al. 2008). ‘Astrophysical’ roughly suggests—over-broadly, at least initially—the types of systems at which the code is aimed, and the kinds of physics and solvers it makes available. ‘Simulation System’ indicates that the code is not a single program, but a collection of modules, structured as classes, that can be invoked by a suitable driver program set up to characterize and initialize a particular problem.

Given that several codes used for astrophysical simulations in general and core-collapse supernova simulations in particular have been described in the literature, an ambitious, new, from-the-ground-up undertaking like GenASiS should offer something new in terms of approach, capability, and/or availability.

The potential generality, extensibility, and maintainability afforded by an object-oriented approach is one such distinguishing feature. This programming paradigm involves several interrelated principles. *Abstraction* identifies the major concepts required by a program, without specifying the details of implementation. *Encapsulation* bundles together and controls access to the data (or *members*) and actions (or *methods*) associated with a particular concept into a self-contained unit—a *class*. (The class itself is just a kind of template or glorified type specification; an *instance* of the class, basically a variable declared with that type specification, is called an *object*. In terms of an analogy with intrinsic data types, a class is to an object as the intrinsic data type `real` is to a variable declared as `real`.) Done well, abstraction and encapsulation lead to *decoupling*—the separation of code into building blocks that are as independent and reusable as possible. Reusability is further enhanced by *polymorphism*, which enables multiple versions or implementations of the same basic concept to be referred to and used interchangeably. Closely related is *inheritance*, which allows new (*child*) classes to be formed from prior (*parent*) classes: existing members and methods are retained or modified, and new members and methods can be added.

Several challenges facing the development and ongoing relevance of complicated simulation codes can be ameliorated by these principles of object-oriented design, and can do so without sacrificing performance. For a problem as rich as core-collapse supernovae, simulating all the desired physics in full dimensionality right off the bat is not feasible; a series of approximations of increasing sophistication must be deployed over time. (Consider for example Newtonian vs. relativistic gravity, mean heavy nucleus composition vs. networks of nuclei, variable Eddington tensor—with any number of closure schemes—vs. Boltzmann transport, and so on.) Even within a particular physics approximation, there are many ways to formulate and solve it. (Imagine different coordinate systems, linear and nonlinear solvers, hydrodynamics reconstruction schemes and Riemann solvers, and so forth.) Multiple hardware platforms (workstations vs. basic clusters vs. increasingly sophisticated supercomputers with heterogeneous, multi-core, accelerator-enhanced nodes), and different software libraries addressing the same needs (e.g. I/O, FFT, solvers, etc.), are available or may come into existence. Some algorithms may be greatly facilitated with the availability of multiple copies of the same basic entities with their own self-contained storage (think for instance of the multiple levels of both meshing and sets of related variables needed by an adaptive mesh refinement scheme—a major motivation for us, as discussed below). All of these challenges are ripe for attack with abstraction, encapsulation, decoupling, polymorphism, and inheritance. Indeed, it is the object-oriented approach that best allows for the flexibility connoted by the ‘General’ in GenASiS—the capacity of the code to include and refer to multiple algorithms, solvers, and physics and numerics choices with the same abstracted names and/or interfaces. Moreover, higher-level organization and flexibility afforded by sophisticated data structures need not sacrifice performance. Lower-level kernels in which the bulk of computational time is spent can be kept isolated and simple—and therefore amenable to compiler optimization—by being written in terms of elementary loops over operations on array variables of intrinsic data types.

Beyond the programming paradigm, one fundamental characteristic of a simulation code that underpins almost everything else is the nature—or even existence, when one considers particle methods—of its meshing, as this is the stage upon which the physics plays out. Smoothed-particle hydrodynamics has been widely used in the broader astrophysics community as an efficient way to get to three dimensions, but its accuracy has been controversial (e.g. Agertz

et al. 2007; Price 2008; Springel 2010b). Unstructured meshes are useful for the complex geometries of engineering contexts, but their high overhead is probably not justified for the more simple geometries of most astrophysics problems. (On the other hand, the relatively new use of adaptive Voronoi tessellations (Springel 2010a) is an interesting new approach that merits further consideration.) Moving patches may be useful for following compact objects in orbit (e.g. Scheel et al. 2006), but introduce additional complicated source terms associated with non-inertial reference frames, and are not an obvious fit for more centrally-condensed problems like core-collapse supernovae. When more structured grid-based approaches are considered, one major choice is whether to use adaptive mesh refinement (AMR) in an effort to deploy computational resources only where needed, and if so, what type of AMR should be used.

Most existing core-collapse supernova codes use strategies other than AMR to handle gravitational collapse. One code uses smoothed-particle hydrodynamics (Fryer et al. 2006). Others use an at least relatively high resolution radial mesh, usually in only one (Rampp and Janka 2002; Thompson et al. 2003; Liebendörfer et al. 2004; Sumiyoshi et al. 2005) or two (Buras et al. 2006; Livne et al. 2007; Bruenn et al. 2009) position space dimensions, often with Lagrangian coordinates (in spherical symmetry) or a moving radial mesh to follow the infall. Use of a radial mesh and associated spherical coordinates imposes severe time step limitations at coordinate singularities unless special measures are taken, such as use of an unstructured mesh to morph to a different coordinate system at low radius (Livne et al. 2007), or an overlap of a radial mesh with a Cartesian mesh at low radius (Scheidegger et al. 2008), or an overlap of two separate radial meshes in a so-called ‘yin-yang’ configuration (Wongwathanarat et al. 2010). While these approaches may give satisfactory gravitational collapse, with a moving radial mesh also reasonably handling the steepening and moving density cliff at the neutron star surface, they cannot address regions of turbulence as well as AMR can. Those codes that do use (or intend to use) AMR use the block-structured variety (Almgren et al. 2010).

That neutrino transport is so integral to the core-collapse supernova explosion mechanism, and yet so overwhelming in its computational demands, has motivated our choice: GenASiS is being developed with an eye towards cell-by-cell AMR (e.g. Khokhlov 1998). Block-structured approaches (Berger and Oliger 1984; Berger and Collela 1989; MacNeice 2000) are more common, but are not necessarily the best choice if neutrino transport is a major focus. One initial motivation for block-structured AMR is the ability to deploy existing ‘unigrid’ time-explicit hydrodynamics solvers on individual regular cell blocks. Moreover, there are some efficiencies associated with the use of predictable basic building blocks. In a multiphysics context, however, cell-by-cell AMR may have some advantages. It is not conceptualized around local time-explicit solvers, so that it is arguably more amenable to elliptic and other global solvers, including those that are time-implicit. It is of course possible to develop global solvers in block-structured AMR—for instance for gravity (e.g. O’Shea et al. 2005; Ricker 2008; Almgren et al. 2010) and radiation (e.g. Rijkhorst et al. 2006; Wise and Abel 2011; Zhang et al. 2011), but it is not clear that this is the most natural environment imaginable for them. Moreover, the computational cost per cell becomes very high if one aims (at least eventually, if not initially) towards the high dimensionality of the full neutrino phase space (position space plus momentum space), and towards very large nuclear reaction networks. In this case cell-by-cell AMR has a potentially significant advantage in requiring a smaller number of total cells for a given accuracy, in part because of the more fine-grained control over cell division and placement, and also because the use of many blocks at a given level of refinement in block-structured AMR leads to a larger ratio of ghost to computational cells.

While a handful of other codes used in astrophysics or cosmology do use cell-by-cell AMR (e.g. Khokhlov 1998; Teyssier 2002; Gittings et al. 2008), our approach in GenASiS has at least one notable difference. In arranging storage and writing solvers, rather than addressing the oct-tree as a single mesh consisting of the union of leaf cells at all levels of refinement, GenASiS has a more explicit level-by-level orientation. This facilitates multigrid approaches to elliptic and time-implicit global solves. Writing solves for single levels, with interactions between levels handled separately, restores some of the flavor of simplicity of the independent solves on individual blocks featured in block-structured AMR; at the same time, treating entire levels at once eliminates the drawbacks of having to stitch together results from

multiple blocks at the same level.

The high-level structure of the core of GenASiS is sketched in Figure 1. Solid lines outline relationships in the source code directory hierarchy, and dashed arrows indicate compilation dependencies. `Modules` and `Programs` are the two highest-level divisions shown. `Modules` comprises the classes forming the central functionality of GenASiS. Each class is defined in a single Fortran module. Fortran programs—drivers that invoke these classes, and begin the execution of some particular computational task—are collected under `Programs`. Both `Modules` and `Programs` contain divisions labeled `Physics`, `Mathematics`, and `Basics`. On the `Modules` side, these categories contain class implementations; dual to these on `Programs` side, in almost complete one-to-one correspondence, are *unit tests* that exercise the capabilities and provide example usage of the individual classes. As indicated by the dashed dependency arrows, the unit tests first depend upon their corresponding classes: `Basics` unit tests depend on `Basics` classes; `Mathematics` unit tests depend on `Mathematics` classes, and through these also on `Basics` classes; and so on. `Programs` also has another division—`Applications`—intended for drivers aimed at purposes beyond unit tests, ranging from the solution of simple physical test problems to the execution of production-scale multiphysics research simulations. These ultimately depend on all `Modules`—`Physics`, `Mathematics`, and `Basics`.

With the top-down overview afforded by Figure 1 in mind, this paper begins our series on GenASiS by describing the fundamental capabilities that underpin implementations of the solvers and physics needed for the simulation of core-collapse supernovae and other astrophysical systems. In particular we discuss the major code division `Basics` and one of the major divisions of `Mathematics`: cell-by-cell refinable `Manifolds`. Two sample problems illustrate our object-oriented approach and exercise the capabilities of the `Basics` and `Manifolds` divisions of GenASiS, including an example with a fixed multilevel grid of a type suitable for gravitational collapse.

## 2. BASICS

The `Basics` division of GenASiS contains some utilitarian functionality for large-scale simulations on distributed-memory supercomputers. (For the place of `Basics` in the overall scheme of GenASiS, see Figure 1.) Its content, as illustrated in the left diagram of Figure 2, includes the divisions `VariableManagement`, `Display`, `MessagePassing`, `FileSystem`, and `Runtime`. Two of these—`Display` and `Runtime`—are framed with boxes of thinner linewidth; these are ‘leaf’ divisions in the sense that they contain no further subdirectories, but only individual files defining classes. The right diagram in Figure 2 shows the structure within `VariableManagement`, which includes the leaf divisions `Specifiers`, `ArrayOperations`, `ArrayArrays`, and `VariableGroups`. The middle right diagram in Figure 2 shows the structure within `MessagePassing`, which includes the leaf divisions `MessagePassingBasics`, `PointToPoint`, and `Collective`. The middle left diagram in Figure 2 shows the structure within `FileSystem`, which includes the leaf divisions `FileSystemBasics`, `GridImageBasics`, `CurveImages`, `StructuredGridImages`, and `UnstructuredGridImages`. In these illustrations the compilation order is from bottom to top; thus the dependencies essentially flow in reverse, from top to bottom.

Before proceeding further we say a bit more about the way in which GenASiS’s classes are arranged and accessed. In discussing the organization of GenASiS, sketched in Figures 1, 2, and similar figures to follow, we refer to the boxed entities as ‘divisions’ of the code. They are not classes, but subdirectories in the source code tree; thus they comprise and compose hierarchically arranged groups of classes. However, even though these divisions are not classes per se, we nevertheless mirror their hierarchical structure in Fortran modules that make groups of classes more convenient to access, as follows. In each directory there is a file defining a Fortran module whose name corresponds to the division or directory, and which contains all the classes (for a leaf division) or groups of classes (for a non-leaf division) within that directory. For instance, in the `Specifiers` directory (see the right diagram in Figure

2) there is a file `Specifiers.f90` defining a module `Specifiers` containing Fortran `use` statements invoking each individual class in that directory. Then the classes in the `ArrayOperations` directory (the next to be compiled) need only the statement `use Specifiers` to access all the classes in that grouping. Similarly, the directory `VariableManagement` contains a file `VariableManagement.f90` as follows:

---

```
1 module VariableManagement
2   use Specifiers
3   use ArrayOperations
4   use ArrayArrays
5   use VariableGroups
6 end module VariableManagement
```

---

Then, in the classes grouped under, for example, `Display` (see the left diagram in Figure 2), the statement `use VariableManagement` suffices to access everything contained within that division.

In this section we illustrate some of the functionality available through `Basics`—and also give a flavor of our use of the object-oriented approach—in the context of a simple example problem.

### 2.1. A Simple Problem and an Object-Oriented Solution

The following tasks define a sample problem we use to give a flavor of our object-oriented approach and illustrate the functionality available through `Basics`: (a) divide a one-dimensional periodic lattice of equidistant points into segments assigned to the several processes of a distributed-memory parallel program; (b) initialize a sine wave whose periodicity is consistent with that of the lattice; (c) exchange ‘ghost’ data points between adjacent lattice segments assigned to different processes; (d) compute the derivative of the sine wave with the familiar second-order centered finite difference formula; (e) evaluate the sine wave and its derivative at a test point; (f) as if for a parameterized phase space type of plot, form coordinate pairs of the sine wave and its derivative for each point; and (g) write output to the file system in a format suitable for visualization. We have contrived a Fortran program (Listing 1) and two supporting modules (Listings 2 and 3) to accomplish these tasks using the GenASiS `Basics` classes. (Obviously this is not a serious application, and for such a simple problem, what follows will seem—like killing a fly with a sledgehammer—as rather more than necessary. But overkill on a simple example now will make later discussions of more sophisticated parts of GenASiS more tractable by illustrating some of the object-oriented mechanisms deployed to more proportionate purpose as the functionality and applicability we discuss widen.) We do not show every detail, but only selected snippets that give a few examples and illustrate some points we wish to make about our object-oriented approach and, in Section 2.2, the functionality available in GenASiS `Basics` classes.

The program outlined in Listing 1 accomplishes the tasks required by our simple problem. The statement `use Basics` in line 3 gives access to all the classes in the code divisions pictured in Figure 2. In line 7 we declare an object `SWD`<sup>1</sup> of class `SineWaveDerivativeForm`,<sup>2</sup> whose definition is available through the `use` statement in line 4. The subroutine calls in lines 11-16 are invocations of methods belonging to the object `SWD`.<sup>3</sup> The comments in

---

<sup>1</sup>The identifier `SWD` is an acronym for `SineWaveDerivative`. Such acronymic naming, especially for objects, is a very common practice of ours.

<sup>2</sup>Our convention is to use the suffix `Form` in class names. We normally define one class per Fortran module, with the suffix preceded by an underscore (`_Form`) in the module name to avoid a collision with the corresponding derived type name.

<sup>3</sup>The syntax `[variable] % [procedure]` to reference methods is new to Fortran 2003. Procedures—either subroutines or functions—are attached to derived types through one of two new mechanisms: `procedure` pointer components, or, as we shall see for instance in Listing 2,



Listing 1 Outline of program `SineWaveDerivative`

---

```

1 program SineWaveDerivative
2
3   use Basics
4   use SineWaveDerivative_Form    !– See Listing 2
5   implicit none
6
7   type ( SineWaveDerivativeForm ) :: SWD    !– See Listing 2
8
9   !– Program initialization omitted; see Section 2.2.5
10
11  call SWD % Initialize ( )           !– Tasks (a) and (b); see Listings 2 and 4
12  call SWD % ExchangeVariables ( )   !– Task (c); see Listings 3 and 9
13  call SWD % ComputeDerivative ( )    !– Task (d); see Listing 3
14  call SWD % EvaluateVariables ( )    !– Task (e); see Listing 3, and Sections 2.2.1, 2.2.2, 2.2.3
15  call SWD % PairVariables ( )        !– Task (f); see Listings 3 and 7, and Section 2.2.1
16  call SWD % WriteVariables ( )       !– Task (g); see Listing 3, and Section 2.2.1
17
18  !– Program finalization omitted
19
20 end program SineWaveDerivative

```

---

these lines correlate the tasks outlined in the previous paragraph with the method invocations.

Evidently we must consider the class `SineWaveDerivativeForm` defined in the module sketched in Listing 2 to begin to see how these tasks are implemented. One member of `SineWaveDerivativeForm` is declared in line 9, the `Amplitude` of the sine wave. It is an object of class `MeasuredValueForm`; as we will discuss in Section 2.2, this allows `Amplitude` to have a unit of measure associated with it. Lines 11–13 are declarations related to two of the methods of `SineWaveDerivativeForm`.<sup>4</sup> The reason for the generic statement in line 12, which provides a publicly available alias `Initialize` for the private method `Initialize_SWD` declared in line 11, will be made clear shortly.

Noting that of the methods invoked in lines 11–16 of Listing 1, only one of them—`Initialize`—is made publicly available (via line 12) in Listing 2, we turn our attention to the indication in line 8 of Listing 2 that this class extends the class `WaveDerivativeTemplate` outlined in Listing 3. The extension of `WaveDerivativeTemplate` to `SineWaveDerivativeForm` is an example of the object-oriented principle of inheritance. The idea in this particular example is that `WaveDerivativeTemplate` contains members and methods useful to solve this simple example problem for any waveform, while `SineWaveDerivativeForm` adds only the elements necessary to specify that the waveform be a sine wave. `SineWaveDerivativeForm` is said to inherit all the members suggested by line 8 of Listing 3, and all the methods declared in lines 10–17. This inheritance is what allows, for instance, the object `SWD` of class `SineWaveDerivativeForm` declared in line 8 of the program in Listing 1 to make the method invocations given in lines 11–16 of that listing.

---

type-bound procedure declarations (e.g. Reid 2007). By default the object that owns the method is the first—and so-called *passed*—argument to the invoked procedure, giving that procedure access to the object’s members. Additional arguments follow in parentheses. In lines 11–16 the parentheses following the method names are empty because the object—in this case `SWD`—is the only argument to those subroutines.

<sup>4</sup>The procedures named in lines 11 and 13 have the `private` attribute, which means they are not accessible outside this module. The default, which we usually affirm explicitly, is `public` availability. These methods also have the `pass` attribute, which affirms the default behavior that the object owning the method is passed as the first argument (e.g. Reid 2007).

Listing 2 Outline of module `SineWaveDerivative_Form`. Used at line 4 of Listing 1.

---

```
1 module SineWaveDerivative_Form
2
3   use Basics
4   use WaveDerivative_Template  !– See Listing 3
5   implicit none
6   private
7
8   type, public, extends ( WaveDerivative_Template ) :: SineWaveDerivativeForm
9     type ( MeasuredValueForm ) :: Amplitude
10  contains
11    procedure, private, pass :: Initialize_SWD  !– See Listing 4
12    generic, public :: Initialize => Initialize_SWD
13    procedure, private, pass :: Waveform
14  end type SineWaveDerivativeForm
15
16 contains
17
18  !– Definitions of subroutine Initialize_SWD and function Waveform omitted
19
20 end module SineWaveDerivative_Form
```

---

Listing 3 Outline of module `WaveDerivative_Template`. Used at line 4 of Listing 2.

---

```
1 module WaveDerivative_Template
2
3   use Basics
4   implicit none
5   private
6
7   type, public, abstract :: WaveDerivative_Template
8     !– Declaration of members omitted
9   contains
10    procedure, private, pass :: Initialize_WD  !– See Listing 5
11    generic, public :: Initialize => Initialize_WD
12    procedure, public, pass :: ExchangeVariables  !– See Listing 9
13    procedure, public, pass :: ComputeDerivative
14    procedure, public, pass :: EvaluateVariables  !– See Sections 2.2.1, 2.2.2, 2.2.3
15    procedure, public, pass :: PairVariables  !– See Listing 7, and Section 2.2.1
16    procedure, public, pass :: WriteVariables  !– See Section 2.2.1
17    procedure ( WaveformInterface ), private, pass, deferred :: Waveform
18  end type WaveDerivative_Template
19
20  !– Declaration of the interface for function WaveformInterface omitted
21
22 contains
23
24  !– Definitions of subroutines Initialize_WD, ExchangeVariables, ComputeDerivative,
25  ! EvaluateVariables, PairVariables, and WriteVariables omitted
26
27 end module WaveDerivative_Template
```

---



Listing 4 subroutine `Initialize_SWD`. Fits in line 18 of Listing 2. Called at line 12 of Listing 1.

---

```
1 subroutine Initialize_SWD ( SWD )
2
3   class ( SineWaveDerivativeForm ), intent ( inout ) :: SWD
4   type ( UnitForm ) :: SineWaveUnit
5
6   SineWaveUnit      = UNIT % IDENTITY
7   SWD % Amplitude = 1.0_KDR * SineWaveUnit
8   !– Statement overriding this default with a value read from parameter file or command line omitted
9
10  call SWD % Initialize ( SineWaveUnit ) !– See Listing 5
11
12 end subroutine Initialize_SWD
```

---

A peculiarity of this particular example is that `WaveDerivativeTemplate` is abstract (as specified in line 7 of Listing 3), as required by the fact that its method `Waveform` is deferred (as declared in line 17 of that listing).<sup>5</sup> This means that, unlike all the other methods declared in lines 10-17 of Listing 3—which are defined or ‘fleshed out’ in this class definition, as suggested by lines 24-25—the method `Waveform` is only given an interface, or specification of its argument list, as suggested by line 20. In this example the method `Waveform` is not fleshed out until `WaveDerivativeTemplate` is extended to `SineWaveDerivativeForm` in Listing 2, in which the previous deferred declaration is said to be *overridden* by the new declaration in line 13, and the method is defined as suggested in line 18. We emphasize that inheritance by type extension does not only occur in going from an abstract class to a non-abstract one, but between any classes at all—and indeed through any number of sequential extensions.

Finally, we note multiple ways in which polymorphism (use of the same name to refer to different variations of the same basic concept) are exhibited in this example.<sup>6</sup> Consider first the *overloading* of the method `Initialize`, which by virtue of the generic statements in line 12 of Listing 2 and line 11 of Listing 3, ends up as an alias for two different subroutines named `Initialize_WD` and `Initialize_SWD`. This is possible because these two routines have different argument lists, which allows the system to *resolve* any particular call (i.e. route it to the correct subroutine) by finding the aliased routine with the appropriate argument list. To see how this works, and to then encounter additional manifestations of polymorphism, consider the subroutines `Initialize_SWD` and `Initialize_WD` in Listings 4 and 5, which would fall in line 18 of Listing 2 of `SineWaveDerivativeForm` and lines 24-25 of Listing 3 of `WaveDerivativeTemplate` respectively. Notice that `Initialize_SWD` has only one argument while `Initialize_WD` has two. This difference allows the call in line 11 of the program in Listing 1 to be resolved to `Initialize_SWD` (with the object `SWD` to the left of the `%` symbol passed as the only argument), and the call in line 10 of Listing 4 to be resolved to `Initialize_WD` (with `SWD` as the first argument and `SineWaveUnit` as the second argument). The declaration of the argument `WD` in line 3 of Listing 5 with the `class` keyword instead of the `type` keyword makes `WD` a polymorphic variable, which means that this argument can be any extension (or extension of extension, and so on) of `WaveDerivativeTemplate`.<sup>7</sup> This is what allows a `SineWaveDerivativeForm` object—ultimately, `SWD` declared in line 7 of Listing 1—to be the first argument of

---

<sup>5</sup>Our convention is to use the suffix `Template` for an abstract class instead of the suffix `Form` we use for typical classes.

<sup>6</sup>Amusingly, the object-oriented principle of polymorphism is itself polymorphic, manifesting itself in multiple ways!

<sup>7</sup>That the `class` keyword denotes a polymorphic variable in Fortran 2003 must not be confused with our use of the term ‘class’ to refer to a construct that bundles together members and methods.

**Listing 5** subroutine `Initialize_WD`. Fits in lines 24-25 of Listing 3. Called at line 10 of Listing 4.

---

```

1 subroutine Initialize_WD ( WD, WaveUnit )
2
3   class ( WaveDerivativeTemplate ), intent ( inout ) :: WD
4   type ( UnitForm ), intent ( in ) :: WaveUnit
5
6   !– Declaration of other variables, and other statements—some of which are in Listings 6 and 8—omitted
7
8   !– X and W are rank-one real arrays storing the lattice point coordinates and wave values on those points respectively
9   W = WD % Waveform ( X ) !– See Listing 2
10
11 end subroutine Initialize_WD

```

---

`Initialize_WD`.

Now for an important point regarding the utility of the object-oriented principles of inheritance and polymorphism for our purposes in GenASiS. The function call in line 9 of Listing 5 of `Initialize_WD` (a method of `WaveDerivativeTemplate` in Listing 3) sets the wave values on the coordinate points (see also the comment in line 8 of Listing 5). As far as `WaveDerivativeTemplate` is concerned this function could return anything; its definition is deferred, with only an interface specified (lines 17 and 20 of Listing 3). But when the argument `WD` is an instance of `SineWaveDerivativeForm`, the function call resolves to the overriding method specified by lines 13 and 18 of Listing 2 of `SineWaveDerivativeForm`, returning values of a sine wave. The point is that the principles of inheritance and polymorphism—exemplified here in the mechanisms of type extension and method overriding—make it much easier to allow lower-level code (in this example, `Initialize_WD` in `WaveDerivativeTemplate`) to access higher-level code (in this example, `Waveform` in `SineWaveDerivativeForm`). This language functionality greatly facilitates, for instance, the separation of the `Mathematics` and `Physics` divisions of GenASiS. For example, solvers for generic classes of equations can be written in `Mathematics`, and then invoked later by a range of different systems whose details are specified in `Physics`. This tremendously enhances the ease and transparency with which one can develop (for example) versatile and widely-applicable solvers.

(The class `UnitForm` appearing in Listings 4 and 5, and Lines 6 and 7 of Listing 4 which give a default setting of the sine wave amplitude, will be discussed shortly in Section 2.2.)

## 2.2. Illustrations of Basics Functionality

With this example program in mind we describe some of the functionality available in `Basics`. In particular we discuss in bottom-up fashion (i.e. in order of compilation) the code divisions appearing in Figure 2.

### 2.2.1. VariableManagement

`VariableManagement` is the first code division appearing in the diagram on the left side of Figure 2. It is one of the `Basics` divisions that is not a leaf division. Its subdivisions `Specifiers`, `ArrayOperations`, `ArrayArrays`, and `VariableGroups` are shown in the rightmost diagram of Figure 2. We discuss each in turn.

**Specifiers** The first division of `VariableManagement` is `Specifiers`, which contains classes used in the specification of number and character variables. This includes some default parameters to be used as `kind` and `len` specifiers for Fortran intrinsic data types. It also includes mathematical and physical constants and a means of dealing with units.

Consider for example the precision and range of numerical variables, which can be specified with a Fortran `kind` parameter. Note for instance the declaration of one of the members of `WaveDerivativeTemplate`,<sup>8</sup> which falls in line 8 in Listing 3:

```
real ( KDR ) :: Wavelength
```

The parameter `KDR` is a nickname, provided by `Specifiers`, for `KIND_DEFAULT % REAL`, one of the members of the object `KIND_DEFAULT` declared and initialized in `Specifiers`.<sup>9</sup> Fortran `kind` parameters also specify the precision and range of explicit numbers, as in the `1.0_KDR` in line 7 of Listing 4. Normally `KIND_DEFAULT % REAL` (and therefore its nickname `KDR`) is set to correspond to double precision; but if one wanted to execute a program in, say, single precision (and assuming all real variables were declared to be of `kind KDR`), changing this single parameter would accomplish the task.

`Specifiers` also provides means for dealing with units, primarily for initialization and output. These means include the classes `MeasuredValueForm` and `UnitForm` and a singleton object `UNIT`. The class `MeasuredValueForm` has a `real` member `Number` and a `character` member `Unit` that specifies a unit of measure (e.g. `'m'` for meters).<sup>10</sup> The class `UnitForm` is meant only for units of measure, and not for typical measured quantities. It is an extension of `MeasuredValueForm` that has an additional `character` member `Label`, the need for which will be explained momentarily. The members of the singleton `UNIT` are objects of `UnitForm` for many units of measure. Now consider one of these members, `UNIT % CENTIMETER`, as an example of the need for the member `Label` of `UnitForm`. The need for this extra member (relative to the members `Number` and `Unit` of the parent class `MeasuredValueForm`) arises from that fact that in its internal computations on plain `real` variables, GenASiS assumes all numbers are given in terms of a single fundamental unit. To be specific, a geometrized system of units is currently employed in which the meter is fundamental. In terms of this fundamental unit,  $1\text{ cm} = 10^{-2}\text{ m}$ . The right-hand side of this expression is reflected in the `MeasuredValueForm` members `Number` and `Unit` of `UnitForm`, i.e. `UNIT % CENTIMETER % Number = 1.0e-2_KDR` and `UNIT % CENTIMETER % Unit = 'm'`; while the `Label` member is associated with the unit in question, i.e. `UNIT % CENTIMETER % Label = 'cm'`. The members `Number` and `Unit` of `UnitForm` are used to convert a number into or out of GenASiS's fundamental unit on input or output, while the `Label` member is used to label the output of a number in the given unit.<sup>11</sup>

---

<sup>8</sup>As a member of `WaveDerivativeTemplate`, at for instance line 6 of Listing 5 this variable would be referenced as `WD % Wavelength`.

<sup>9</sup>By convention, when we write a class name that is not an acronym in all caps (for instance `KIND_DEFAULT`) it refers to a *singleton*, a special type of object intended to be the only instance of that class ever declared. When we write a class member (e.g. the `REAL` in `KIND_DEFAULT % REAL`) in all caps it denotes a parameter that is intended to remain unchanged throughout program execution. All-caps class members remaining unchanged is at the mercy of convention, except in the case of a singleton object declared with the `parameter` attribute, in which case all of its members automatically remain unchanged.

<sup>10</sup>Our apologies for using 'unit' in three ways here: first, as the name of a `character` member of `MeasuredValueForm`; second, in the name of the class `UnitForm`; and third, as the name of the singleton object `UNIT`. Which sense is meant in a particular expression should be clear in context.

<sup>11</sup>We emphasize that type extension via the Fortran 2003 `extends` keyword (as in line 8 of Listing 2) gives the extended type direct access to the members of the parent type. For example, if `CoordinateUnit` is declared to be of type `UnitForm`, we can refer directly to the members `CoordinateUnit % Number` and `CoordinateUnit % Unit` inherited from the parent type `MeasuredValueForm`, in addition to the member `CoordinateUnit % Label` added to the extended type.

Listing 6 Coordinate and wavelength initialization in `Initialize_WD`. Fits in line 6 of Listing 5.

---

```

1  type ( UnitForm ) :: CoordinateUnit
2  !– Other declarations and statements omitted
3
4  CoordinateUnit      = UNIT % CENTIMETER
5  WD % MinCoordinate = 0.0_KDR * CoordinateUnit
6  WD % MaxCoordinate = 1.0_KDR * CoordinateUnit
7  !– Statement overriding these defaults with values read from a parameter file or command line omitted
8  !– Display to screen omitted; see Section 2.2.2
9
10 WD % Wavelength = ( WD % MaxCoordinate – WD % MinCoordinate ) / WD % nWavelengths

```

---

We are now in a position to see how initializations with units can be coded directly. (Examples of initialization with units when reading numbers from parameter files or as command line options, and of output with units to the screen or to a file, will be discussed in connection with the `FileSystem`, `Runtime`, and `Display` portions of `Basics`.) Here is the declaration of two other members of `WaveDerivativeTemplate`, which falls in line 8 in Listing 3:

```
type ( MeasuredValueForm ) :: MinCoordinate , MaxCoordinate
```

Listing 6, which would fall at line 6 of Listing 5 of subroutine `Initialize_WD`, sets the default endpoints and units of the periodic lattice, and uses these endpoints (together with an integer member `nWavelengths` specifying the number of wavelengths of the periodic waveform to set on the lattice) to set the wavelength. The arithmetic and assignment operations involving derived types in lines 5 and 6 of Listing 6 behave in expected ways: the explicit real number multiplies the `Number` member on the right-hand side, and the `Label` member in the `UnitForm` on the right-hand side is simply ignored in the assignment to the `MeasuredValueForm` on the left-hand side.<sup>12</sup> Multiplication of an explicit number by one of the `UNIT` members puts the coordinates of the lattice endpoints into GenASIS’s fundamental working unit, so that one can assign their difference to the plain real member `Wavelength` in line 10. (In the assignment of a `MeasuredValueForm` to a real, the `Unit` member is simply ignored.) Another example is lines 6 and 7 of Listing 4, in which the member `Amplitude` of `SineWaveDerivativeForm` (declared at line 9 of Listing 2) receives a default initialization to a dimensionless value of unity.

Finally, we note that mathematical and physical constants can be accessed through the singleton `CONSTANT`. For instance, the number  $\pi$  is referenced in the expression

```
2.0_KDR * CONSTANT % PI / WD % Wavelength
```

for the wavenumber in function `Waveform` (whose definition is noted at line 18 of Listing 2 for `SineWaveDerivativeForm`).

**ArrayOperations** The second division of `VariableManagement` is `ArrayOperations`, which includes some basic operations on arrays. Some of these—such as `Clear` and `Copy`—are merely overloaded names for subroutines whose arguments are intrinsic data types stripped of any potentially complicating contexts or attributes that might inhibit compiler optimization.<sup>13</sup> The purpose of these wrappers is to expose elementary variables

---

<sup>12</sup>These and like results are achieved by overloading of the arithmetic and assignment operators.

<sup>13</sup>The modules defining these two operations are `Clear_Command` and `Copy_Command`. We use the suffix `Command` instead of `Form` in naming a module when it is more convenient to simply implement an action, as opposed to a class with both members and methods. Slavish

to the compiler so that it might use fast methods (such as a system call to set values in memory) it might not otherwise employ if, say, the array has the `pointer` attribute, is deeply embedded in a complicated expression involving a sequence of derived type member references, etc. Other array operations have a more substantial purpose, such as `Sort` and `Search`.<sup>14</sup> The `Search` operation, for instance, is used in task (e) of our sample problem (evaluate the sine wave and its derivative at a test point). The method `EvaluateVariables` of `WaveDerivativeTemplate` includes the line

```
call Search ( Coordinate , X_Evaluate , iE )
```

The first two arguments are inputs: `Coordinate` is a rank-one real array containing the coordinates of the lattice points, and `X_Evaluate` is the real coordinate of the test point at which the sine wave and its derivative are to be evaluated. Upon return, the last argument `iE` is set to the index of `Coordinate` for which `Coordinate ( iE ) <= X_Evaluate < Coordinate ( iE + 1 )`. The search having bracketed the test point between array indices `iE` and `iE + 1`, the coordinate and sine wave and derivative values at these points can then be used to perform the desired evaluations by linear interpolation.

**ArrayArrays** The next division of `VariableManagement` is `ArrayArrays`, which contains classes that can be used to form arrays of arrays. One application is the construction of so-called *ragged arrays*. In the rank-two case a ragged array can be conceptualized (following the Fortran row-major perspective) as a matrix whose columns have different numbers of elements. This can be useful when one wants an indexed collection of rank-one arrays that are not all the same length. Even if they are all the same length, one might want to conceptualize the data primarily as a collection of rank-one objects rather than as a rank-two array. This is the way our sample program handles task (f)—set up coordinate pairs of the sine wave and its derivative at each point, as if for a phase space type of plot. The method `PairVariables` of `WaveDerivativeTemplate` is outlined in Listing 7. We want one variable pair—a value of the wave paired with a value of the derivative—for each coordinate point. Accordingly we declare a rank-one array `VariablePair` of `ArrayReal_1D_Form` (line 7) and allocate it to have the same number of elements as the array of coordinate values (line 10). The class `ArrayReal_1D_Form` has an allocatable rank-one real array member `Value`. In line 12, calling the `Initialize` method of each element of the array `VariablePair` with the argument 2 causes the member `Value` to be allocated to be two elements long.<sup>15</sup> Then the wave and its derivative at a single point are assigned to the member `Value` of one of the `VariablePair` elements in line 13. Thus we have formed a rank-one array (one for each `Coordinate` point; lines 7, 10) of rank-one arrays of length 2 (for wave/derivative coordinate pairs; lines 12-13). As an aside, we note that the method `Initialize` of `ArrayReal_1D_Form` is overloaded; possible alternative arguments include an array shape (used here), an explicit array to be copied, or another `ArrayReal_1D_Form` to be copied. For instance, the single line

```
call VariablePair ( iP ) % Initialize ( [ Wave ( iP ), Derivative ( iP ) ] )
```

---

adherence to object-oriented principles would implement such a case as a class with a (perhaps overloaded) method and no members. In the case of pure actions this is more cumbersome than necessary, leading us to revert to a more procedural rather than object-oriented perspective for `Commands`. Therefore the mechanism of overloading in this case is not the Fortran 2003 `generic` statement used in connection with type-bound procedures; instead it is the use, possible since Fortran 90, of an `interface` block to define a generic name for a set of procedures. Nevertheless the basic idea of the overloading is still the same, i.e. use of a single name to refer to several possible routines, distinguished by and resolved according to their distinct argument lists.

<sup>14</sup>These are defined in the modules `Sort_Command` and `Search_Command`.

<sup>15</sup>We emphasize that the method `Initialize` of `ArrayReal_1D_Form` invoked in line 12 is completely distinct and separate from the `Initialize` method of `WaveDerivativeTemplate` and its extension `SineWaveDerivativeForm` we have seen previously.

Listing 7 subroutine `PairVariables`. Fits in lines 24-25 of Listing 3.

---

```

1 subroutine PairVariables ( WD )
2
3   class ( WaveDerivativeTemplate ), intent ( inout ) :: WD
4
5   integer ( KDI ) :: iP  !- iPair
6   !- Declaration related to output to screen omitted
7   type ( ArrayReal_1D_Form ), dimension ( : ), allocatable :: VariablePair
8
9   !- Coordinate, Wave, and Derivative are rank-one arrays
10  allocate ( VariablePair ( size ( Coordinate ) ) )
11  do iP = 1, size ( VariablePair )
12    call VariablePair ( iP ) % Initialize ( 2 )
13    VariablePair ( iP ) % Value = [ Wave ( iP ), Derivative ( iP ) ]
14    !- Output to screen omitted
15  end do
16
17 end subroutine PairVariables

```

---

could have been used instead of lines 16-17. In addition to `ArrayReal_1D_Form` there are of course other analogous classes for arrays of other types and ranks.

**VariableGroups** The final division of `VariableManagement` is `VariableGroups`, which contains some infrastructure we use extensively in handling collections of variables, especially sets of related physical fields. In particular, the class `VariableGroupForm` includes both metadata about the variables (names, units, etc.) and storage for the variable data itself. The data storage is a rank-two real array member `Value` of `VariableGroupForm`; the first dimension indexes separate points, and the second dimension indexes individual variables.

Our sample program uses an instance of `VariableGroupForm` to store the variables on its lattice points. To prepare to understand its initialization we note the declarations of some other members of `WaveDerivativeTemplate`, which fall in line 8 in Listing 3:

```

integer ( KDI ) :: COORDINATE = 1, WAVE = 2, DERIVATIVE = 3, N_VARIABLES = 3, nPointsLocal
type ( VariableGroupForm ) :: VariableGroup

```

Member `VariableGroup` will store three variables, hence `N_VARIABLES = 3`. To avoid the confusing appearance of plain numerical indices, arrays over these variables will be indexed by the members `COORDINATE`, `WAVE`, and `DERIVATIVE`, as will be made more clear shortly. The integer `nPointsLocal` stores the number of local lattice points assigned to each segment of the distributed lattice (recall that the lattice as a whole is divided among the processes of a distributed-memory program). Listing 8, which would fall at line 6 of Listing 5 of subroutine `Initialize_WD`, shows the initialization of the `VariableGroup` member. Lines 1 and 2 of Listing 8 initialize local variables used to prepare the variable metadata, i.e. the units and names of the three variables to be stored in member `VariableGroup`. This metadata is then prepared in lines 5-7 and 9-11. (We have already seen the declaration of `WaveUnit` and `CoordinateUnit` as instances of `UnitForm` in previous listings.<sup>16</sup>) The ini-

---

<sup>16</sup>Note the division of one instance of `UnitForm` by another in the right-hand side of line 7. We have overloaded the arithmetic operators in such a way that arithmetic can be performed on instances of `MeasuredValueForm` and `UnitForm`, including some basic handling of the consequences for their `character` members expressing units of measure.

Listing 8 Variable group initialization in `Initialize_WD`. Fits in line 6 of Listing 5.

---

```

1  type ( UnitForm ), dimension ( WD % N_VARIABLES ) :: VariableUnit
2  character ( LDL ), dimension ( WD % N_VARIABLES ) :: VariableName
3  !– Other declarations and statements omitted
4
5  VariableUnit ( WD % COORDINATE ) = CoordinateUnit
6  VariableUnit ( WD % WAVE )       = WaveUnit
7  VariableUnit ( WD % DERIVATIVE ) = WaveUnit / CoordinateUnit
8
9  VariableName ( WD % COORDINATE ) = 'Coordinate'
10 VariableName ( WD % WAVE )       = 'Wave'
11 VariableName ( WD % DERIVATIVE ) = 'Derivative'
12
13 call WD % VariableGroup % Initialize &
14     ( [ WD % nPointsLocal + 2, WD % N_VARIABLES ], &
15       VariableOption = VariableName, UnitOption = VariableUnit, ClearOption = .true. )

```

---

tialization of the `VariableGroup` member then comes in the call in line 13 to the `Initialize` method of `VariableGroupForm`. The first argument is an array with two elements specifying the shape of the data storage: the first element gives the number of points (the number of points in the local lattice segment plus two ghost points), and the second element specifies the number of variables. The optional arguments that follow specify the aforementioned variable metadata, and instruct that the data array should be ‘cleared’ (initialized to 0.0).

It is often convenient to use aliases to access the variable data in an instance of `VariableGroupForm`. Consider for instance the rank-one arrays `Coordinate`, `Wave`, and `Derivative` referred to in Listing 7 of the method `PairVariables` of `WaveDerivativeTemplate`. The following Fortran 2003 (nested) `associate` constructs<sup>17</sup> placed at line 9 in Listing 7 arranges these aliases for use as rank-one arrays:

```

associate ( VG => WD % VariableGroup )
associate &
  ( Coordinate => VG % Value ( :, WD % COORDINATE ), &
    Wave       => VG % Value ( :, WD % WAVE ), &
    Derivative => VG % Value ( :, WD % DERIVATIVE ) )

```

An appropriate number of `end associate` statements should then follow line 15.

A `VariableGroupForm` can have its own storage, as in the above example; or point to existing storage, perhaps for the purpose of selecting only a subset of variables from an existing variable group. For instance, in method `WriteVariables` of `WaveDerivativeTemplate` the two dependent variables (the wave and its derivative) are treated differently from the independent variable (the coordinate of the lattice points). Thus in `WriteVariables` a new instance of `VariableGroupForm` is set up as a local variable:

```

type ( VariableGroupForm ) :: VG_Curves

call VG_Curves % Initialize &
  ( WD % VariableGroup, SelectedOption = [ WD % WAVE, WD % DERIVATIVE ] )

```

Again, overloading of the `Initialize` method allows variation in behavior. In this case the local object

---

<sup>17</sup>This aliasing mechanism, new to Fortran 2003, avoids the need to declare pointer variables to be used as shorthand names and allows the compiler to produce more efficient code.



`VG_Curves` uses the member `VariableGroup` of an instance of `WaveDerivativeTemplate` as the basis of its initialization, and in fact will point to its existing data storage rather than allocate its own and perform a copy operation. But as the second argument suggests, only our dependent variables are ‘selected’ in the new variable group `VG_Curves`. Code written to use objects of `VariableGroupForm`—such as the GenASiS I/O classes invoked by method `WriteVariables` of our sample program—will then act only on the ‘selected’ variables.

One may wonder why the trouble should be taken to use a class like `VariableGroupForm` rather than work directly with explicitly declared arrays of variables. The reason is that this simple construct greatly simplifies and renders more generic many tasks. For instance, as just mentioned, the file system facilities described later are written in terms of variable groups, so that a user need not program such operations in detail for the many sets of variables that may arise in different problems. Another example is the exchange of ghost cell data in multidimensional manifolds (Section 3), which is also handled in terms of variable groups. In writing versatile and reusable code it is much better to be able to simply set up a variable group rather than have to continually rewrite (for example) explicit ghost exchange or I/O library calls for the many different variables that may appear in, or be added to or removed from, various applications.

Finally, we mention that the code division `VariableGroups` also includes a class `PackedVariableGroupForm` used to collect selected rows and columns of a `VariableGroupForm` into contiguous memory for more efficient processing.

### 2.2.2. Display

After `VariableManagement` comes the leaf division `Display` (see the diagram on the left side of Figure 2), which contains infrastructure for displaying messages and variables to the standard output in a uniform and orderly way. The two central features available through `Display` are the singleton object `CONSOLE`, which has members that are used to control output to the standard display (‘`stdout`’); and the heavily overloaded routine `Show`,<sup>18</sup> which manages the hassles of displaying various data types with their respective formatting.

One issue that arises in a distributed-memory parallel computing environment is that, absent intervention by the programmer, output emerges from all processes in an asynchronous manner. This typically results in unordered output of unwanted redundancy. The use of `if` statements can avoid this by allowing only a particular process to output to the screen, but a proliferation of `if` statements with every `print` statement becomes tedious (and difficult to change for debugging purposes if a global variable specifying the process to be displayed is not used). Thus `CONSOLE` has a member `DisplayRank` used by the `Show` command to display output only from the process of the chosen rank, without the programmer having to continually supply `if` statements. If needed, the member `DisplayRank` can be changed anywhere in the program. Consider for instance task (e) in our sample program (evaluate the sine wave and its derivative at a test point), implemented by the method `EvaluateVariables` of `WaveDerivativeTemplate`. In general the test point will not fall in the lattice segment belonging to the default setting of `DisplayRank`. Assuming all processes are aware of `EvaluateRank` (the rank of the process owning the relevant lattice segment), control of the display can be given to that process as follows:

```
DisplayRankOld = CONSOLE % DisplayRank
call CONSOLE % SetDisplayRank ( EvaluateRank )
!– Perform function evaluations and display results
call CONSOLE % SetDisplayRank ( DisplayRankOld )
```

---

<sup>18</sup>Defined in the module `Show_Command`.

The second invocation of the `SetDisplayRank` method returns control of the display to the process that previously had it. Alternatively, instead of calling `SetDisplayRank` one can supply an optional argument to the `Show` command specifying, for that call only, which process should be visible.

Another issue related to output is that different pieces of information are of different levels of interest. In a production run one might want minimal output, while more verbose output may be useful for debugging. `CONSOLE` has a member `Verbosity` that addresses this need. It can be set to one of several levels denoted, in descending order of importance, by the `CONSOLE` members `ERROR`, `WARNING`, `INFO_1`, . . . , `INFO_7`. In a call to `Show` one of these same members can be given as an optional argument; if the importance is thus tagged as less than `Verbosity`, the output is suppressed. In this optional argument's absence an importance of `INFO_1` is assumed.

The `Show` command provides a very simple interface for a wide range of formatted and labeled output. There are provisions for intrinsic data types, objects of `MeasuredValueForm`, and arrays of all these. An object of `UnitForm` can be supplied to modify the units (including the unit label itself) in which real or `MeasuredValueForm` numbers are displayed. Consider for instance the following lines, which fit in line 8 of Listing 6:

```
call Show ( WD % MinCoordinate , 'MinCoordinate' )
call Show ( WD % MaxCoordinate , 'MaxCoordinate' )
call Show ( WD % MinCoordinate , CoordinateUnit , 'MinCoordinate' )
call Show ( WD % MaxCoordinate , CoordinateUnit , 'MaxCoordinate' )
```

Recall that the `MinCoordinate` and `MaxCoordinate` members are of `MeasuredValueForm`, and that their assignment in lines 5-6 of Listing 6 puts them into GenASiS's internal geometrized units—in this case, meters. This means that the first two `Show` calls above will give output in this unit, tagged with the unit string 'm', and labeled with the strings 'MinCoordinate' and 'MaxCoordinate'. The output from the last two calls, which supply an additional argument `CoordinateUnit` (set to `UNIT % CENTIMETER` in line 4 of Listing 6), will be appropriately converted and tagged with the unit string 'cm'. The output will only come from the process specified by `CONSOLE % DisplayRank`, and the absence of the optional argument specifying importance means they will be regarded as `INFO_1`. As another example consider the line

```
call Show ( Derivative , DerivativeUnit , 'Derivative' , CONSOLE % INFO_3 )
```

Here `Derivative` is an alias to the array of the wave derivative values. `DerivativeUnit` is an alias to the value set in line 7 of Listing 8; with the defaults of a dimensionless `WaveUnit` (see Listing 4) and `CoordinateUnit` of centimeters in our sample program, `DerivativeUnit` turns out to be inverse centimeters. Thus this `Show` call will output each value of the array of derivative values on a separate line, labeled with the unit string 'cm<sup>-1</sup>'. The output will only appear from the process of rank `CONSOLE % DisplayRank`, and only if `CONSOLE % Verbosity` is high enough to include `INFO_3`.<sup>19</sup>

### 2.2.3. MessagePassing

The next division of `Basics` is `MessagePassing` (see the diagram on the left side of Figure 2), which contains some classes that abstract the data and methods useful for working in a message passing parallel computing environment. As shown in the middle right diagram in Figure 2, it has three subdivisions: `MessagePassingBasics`,

---

<sup>19</sup>The setting of `CONSOLE % Verbosity` for a program is done via a user-specified parameter file (see Section 2.2.4) and/or command-line arguments (see Section 2.2.5).

PointToPoint, and Collective.

Before describing the leaf divisions under `MessagePassing` we note that one may ask why we bother to create and use message passing classes when MPI (Message Passing Interface) (Gropp et al. 1999) is well-established, widely used, and seems likely to remain for many years the dominant library for message passing in distributed-memory applications. Our message passing classes do indeed use MPI ‘under the hood,’ but in principle could be given other back ends without modifying their references in higher-level code. Our interfaces are somewhat streamlined relative to direct MPI calls while providing methods for common use cases relevant to our system, and thus provide a simpler experience for the user. In terms of object-oriented design patterns (Gamma et al. 1994), our `MessagePassing` classes provide a ‘façade’ that shields the user from concerns such as the kind and size of data types being sent and received (e.g. single precision vs. double precision, and beyond, in the case of `real` data) and declaring many variables—handles, buffers, tags, and so on—for use in MPI calls. In our approach many of these needed variables are encapsulated in the classes we define. Finally, we note that the MPI library is mostly procedural in orientation and uses Fortran 77 semantics for its routines, sometimes requiring more arguments than necessary if the equivalent modern Fortran interfaces were available. Our interfaces relieve the user of these burdens.

**MessagePassingBasics** The most important class to mention here is `CommunicatorForm`. Inspired (obviously) by the notion of an MPI communicator, this class connects a group of processes in a distributed-memory parallel program. The members of `CommunicatorForm` include its `Handle`, `Size`, and `Rank`. Its method `Initialize` is overloaded to provide for either starting the message passing system, i.e. initializing MPI and thus creating the ‘global’ or ‘world’ communicator; or for creating a ‘subcommunicator’ that connects only a subset of the processes connected by a parent communicator. The method `Synchronize` pauses all processes until they reach the call to this method (i.e. an MPI ‘barrier’). For instance, recall the discussion in the second paragraph of Section 2.2.2 on changing the `DisplayRank` during the execution of task (e) (evaluate the sine wave and its derivative at a test point). To ensure the output from two different processes remains ordered (and with the alias `C` for the `CommunicatorForm` object representing the global communicator), the line

```
call C % Synchronize ( )
```

is included before each of the two calls to the `SetDisplayRank` method. Another method of `CommunicatorForm` is `Abort`, which provides an emergency exit from distributed-memory program execution. Normal termination of the message passing system (i.e. finalization of MPI) occurs when the `CommunicatorForm` object representing the global communicator goes out of existence.

**PointToPoint** A single message from one process to another is a ‘point-to-point’ communication. The division `PointToPoint` provides classes for single incoming and outgoing messages of different intrinsic data types, as well as for arrays of such messages. These are used in task (c) of our sample program, as shown in Listing 9 for the method `ExchangeVariables` of `WaveDerivativeTemplate`. The exchange of ghost point data requires each process to receive two messages and send two messages, one each to the processes owning the ‘previous’ and ‘next’ segments of the lattice (the ranks of these processes are set in lines 11 and 12). Lines 6 and 7 declare objects `IMA` and `OMA` of classes `IncomingMessageArrayRealForm` and `OutgoingMessageArrayRealForm` respectively. In this case the initializations of `IMA` and `OMA` (lines 14 and 17) are identical. The first argument is an object of `CommunicatorForm`, here aliased as `C`. That the subsequent arguments are arrays of length 2 indicates that `IMA` and `OMA` will each have two messages. The second argument is an array of tags; in this case these are trivial because the fact that only one message will arrive from a given process means that there is no need to more specifically label (i.e. tag) them. The third argument is an array of the ranks of the process ranks from (to) which the two messages are

Listing 9 subroutine `ExchangeVariables`. Fits in lines 24-25 of Listing 3. Called at line 13 of Listing 1.

---

```
1 subroutine ExchangeVariables ( WD )
2
3   class ( WaveDerivativeTemplate ), intent ( inout ) :: WD
4
5   integer ( KDI ) :: PreviousRank , NextRank
6   type ( IncomingMessageArrayRealForm ) :: IMA
7   type ( OutgoingMessageArrayRealForm ) :: OMA
8
9   ! – Aliasing via associate construct omitted
10
11   PreviousRank = modulo ( myRank – 1, nRanks )
12   NextRank     = modulo ( myRank + 1, nRanks )
13
14   call IMA % Initialize ( C, [ 0, 0 ], [ PreviousRank , NextRank ], [ 2, 2 ] )
15   call IMA % Receive ( )
16
17   call OMA % Initialize ( C, [ 0, 0 ], [ PreviousRank , NextRank ], [ 2, 2 ] )
18   OMA % Message ( 1 ) % Value = [ Coordinate ( iF ), Wave ( iF ) ]
19   OMA % Message ( 2 ) % Value = [ Coordinate ( iL ), Wave ( iL ) ]
20   call OMA % Send ( )
21
22   call IMA % WaitAll ( )
23   Coordinate ( iIG ) = IMA % Message ( 1 ) % Value ( 1 )
24   Wave       ( iIG ) = IMA % Message ( 1 ) % Value ( 2 )
25   Coordinate ( iOG ) = IMA % Message ( 2 ) % Value ( 1 )
26   Wave       ( iOG ) = IMA % Message ( 2 ) % Value ( 2 )
27
28   call OMA % WaitAll ( )
29
30   ! – Display to screen omitted
31
32 end subroutine ExchangeVariables
```

---

to be received (sent). The fourth argument is an array containing the numbers of elements in each message; this is 2 for each message because we are sending two values for each ghost point, i.e. a coordinate value and a wave value. Non-blocking ‘receives’ of the incoming messages are posted by the single call in line 15. Non-blocking ‘sends’ are initiated (line 20) after the outgoing message buffers to the previous and next processes are loaded (lines 18 and 19) with data from the first (indexed by alias `iF`) and last (indexed by alias `iL`) points in the segment. The `WaitAll` method (line 22) ensures that the data have been received and are available in the incoming message buffers before they are stored in the inner (indexed by alias `iIG`) and outer (indexed by alias `iOG`) ghost points (lines 23-26). A perfunctory wait for the outgoing messages to finish sending (line 28) completes the exchange.

**Collective** A communication that involves a group of processes is said to be ‘collective’. The division `Collective` contains classes that provide for collective communications involving different intrinsic data types. The collective communication class for a given data type is generic in the sense that its members are sufficient for broadcast, gather, scatter, and combined scatter-gather<sup>20</sup> communications, and for collection operations (i.e. reductions of data); thus all of these are simply implemented as methods of a single class. For the gather and reduction methods, the absence or presence of the optional argument `RootOption` in the initialization of a collective communication object determines whether or not the gathered or reduced value is distributed to all processes.<sup>21</sup> This option should also be used in connection with broadcast and scatter calls: by definition, these communications involve a ‘root’ process that is the source of data to be sent to all the processes in the group. As an example of a collective communication in our sample program, the `EvaluateVariables` method of `WaveDerivativeTemplate` makes all processes aware, via a gather operation, of which process owns the test point and will perform the function evaluations (task (e)); see also Section 2.2.1, “ArrayOperations”; and Section 2.2.2). Consider the following lines:

```
call CO.Evaluate % Initialize ( C, nOutgoing = [ 1 ], nIncoming = [ C % Size ] )
CO.Evaluate % Outgoing % Value = [ myEvaluate ]
call CO.Evaluate % Gather ( )
EvaluateRank = maxloc ( CO.Evaluate % Incoming % Value, dim = 1 ) - 1
```

The object `CO.Evaluate` is an instance of the class `CollectiveOperationIntegerForm`. The first argument in its initialization in the first line is an object of `CommunicatorForm`, aliased here as `C`. The second and third arguments in the initialization specify how large the outgoing and incoming buffers need to be. In the second line, the integer `myEvaluate` assigned to the outgoing buffer on each process has been set to 0 on all processes except the one owning the test point, which sets it to 1. The gather operation is invoked in the third line; the absence of the optional argument `RootOption` in the initialization means that the gathered value is distributed to all processes. Then all processes can determine `EvaluateRank` by finding the index of the incoming buffer element that is nonzero.

#### 2.2.4. *FileSystem*

As seen in the diagram on the left side of Figure 2, the next division of `Basics` is `FileSystem`, whose classes handle I/O to disk. The middle left diagram in Figure 2 shows divisions `FileSystemBasics`, `GridImageBasics`, `CurveImages`, `StructuredGridImages`, and `UnstructuredGridImages`. The major classes in `FileSystem` used to create and interact with files of various kinds have the term ‘stream’ in their names, connoting a source or sink of data flow. The names of classes used to store grids and data intended for visual-

---

<sup>20</sup>‘AllToAll’ routines in MPI.

<sup>21</sup>In MPI this corresponds to `MPI_Allgather` vs. `MPI_Gather`, or `MPI_Allreduce` vs. `MPI_Reduce`.

ization and/or checkpoint/restart include the term ‘image,’ which evokes both colloquial and computing connotations (respectively ‘picture’ and ‘a computer file capturing the contents of some computing element—hard drive, system memory, etc.’).

**FileSystemBasics** In addition to some lower-level functionality, the classes in this division include two kinds of streams.

`ParametersStreamForm` is used to read scalar or one-dimensional array parameters of various datatypes from a text file, each line of which contains an entry of the form `[name]=[value]` or `[name]=[value1], [value2], ...`. Units may be attached to real values with a tilde (~) followed by an all-caps string denoting one of the units available in `UNIT_Singleton` (see “Specifiers” in Section 2.2.1). For example, the default dimensionless value 1.0 of the `Amplitude` member of `SineWaveDerivativeForm` (see Listings 2 and 4) could be overwritten with an entry like

```
Amplitude=5.0~TESLA
```

in a parameter file. Parameters need not appear in the file in the order searched for by the program; indeed the absence of a parameter, or even the absence of the parameter file, will not in themselves cause a program exit or crash (though the lack of sensible default values for absent parameters certainly might). Our example program `SineWaveDerivative` does not directly invoke objects of `ParametersStreamForm`, but instead uses a higher-level parameter search facility described in Section 2.2.5 that looks both in a parameter file and at command line options.

A second type of stream in `FileSystemBasics` is `TableStreamForm`, which reads simple two-dimensional tables of data from a text file.

**GridImageBasics** The remaining divisions in `FileSystem`, beginning with `GridImageBasics`, provide a façade facilitating interaction with sophisticated I/O libraries. Different libraries may be implemented as ‘back ends’, but the only specific implementation at present is the Silo library,<sup>22</sup> which is convenient for use with the visualization package `VisIt`.<sup>23</sup>

The major class of interest in this division is `GridImageStreamForm`, which handles some basic aspects of I/O that are the same regardless of the type(s) of grid(s) to be written. Its members and methods allow one to, for instance, open (for reading or writing) or close a file; make a directory (virtual within a file or actual in the file system, which may depend on the I/O library back end); list an existing file’s contents; and automatically track filename numbering for consecutive write operations (e.g. for periodic output in time-dependent evolution). In our example program, the method `WriteVariables` of `WaveDerivativeTemplate` declares and initializes an instance of `GridImageStreamForm`—and uses that object to open a file—as follows:

```
type ( GridImageStreamForm ) :: GIS

call GIS % Initialize ( Name, CommunicatorOption = C )
call GIS % Open ( GIS % ACCESS_CREATE )
```

---

<sup>22</sup><https://wci.llnl.gov/codes/silo>

<sup>23</sup><https://wci.llnl.gov/codes/visit>

Here `Name` and `C` are aliases for the program’s name and its global communicator. The argument used in calling the `Open` method specifies the mode of access to the stream object. In this example, we ask the stream object to create a new file. With this access mode, a file number counter associated with the stream object is incremented and appended to the `Name` argument to form the filename, and then a new file is created with this filename. This access mode also implies read and write access to the newly created file. Other possible access modes are `ACCESS_WRITE`, which opens an existing file for writing by appending new data to the end of the file; and `ACCESS_READ`, which opens an existing file for reading only.<sup>24</sup> After the data is written, the call

```
call GIS % Close ( )
```

flushes the data associated with the stream<sup>25</sup> and closes the file.

**CurveImages** The class `CurveImageForm` is used write a one-dimensional grid and data on it, which can be used to generate curves in a Cartesian  $xy$  plane. In our sample program the wave and its derivative (which can be used as dependent variables in curve plots) and the spatial coordinates of the lattice points (which can be used as the independent variable) are written with the following lines, which (aside from the declaration of the `CurveImageForm` object `CI`) lie between the calls to `GIS % Open` and `GIS % Close` just mentioned above:

```
type ( CurveImageForm ) :: &
  CI

call CI % Initialize ( GIS )
call CI % SetGrid &
  ( Directory = 'Curves', NodeCoordinate = Coordinate, nProperCells = WD % nPointsLocal, &
    oValue = 0, CoordinateUnitOption = CoordinateUnit )
call CI % AddVariableGroup ( VG.Curves )
call CI % Write ( )
```

The `GridImageStream` object `GIS` mentioned above is needed as an argument to the `CI % Initialize` method. The call to `CI % SetGrid` specifies a directory, gives the spatial coordinates of the lattice points, and includes an optional argument specifying the units of those coordinates. The variable group `VG.Curves`, which comprises only the data of the wave and its derivative, was discussed in the “VariableGroups” portion of Section 2.2.1. Its use as an argument to the `CI % AddVariableGroup` method means that the variable data and metadata (names, units, etc.) specified by the variable group are available to the `CurveImageForm` methods and can be included in the file as supported by the I/O library back end, which in turn allows a visualization package that supports the format to display it directly. Here we attach only a single variable group, but any number can be added. With these preparations, all that is needed is the call to `CI % Write`. Plots of the saved data—in which the previously specified units and variable names are automatically used by the visualization package `VisIt` without further intervention by the user—are shown in Figure 3.

**StructuredGridImages, UnstructuredGridImages** The classes `StructuredGridImageForm` and `UnstructuredGridImageForm` are used to write two- or three-dimensional grids and data on them, which can be used to generate various kinds of multidimensional plots. A structured grid may be rectilinear or curvilinear, but is in either case ‘logically rectangular’: the quadrilateral or cuboid cells filling the domain can be indexed with a pair

---

<sup>24</sup>These access modes are parameter class members of the stream object (see also <sup>9</sup>).

<sup>25</sup>Depending on the I/O library and hardware, a call to the write method may only buffer the data and not actually write to the file system until the stream is closed.



or triplet of integers, with a single list of coordinates in each dimension sufficing (together with the integer values of an index pair or triplet) to specify the edge values of any particular cell. In an unstructured grid—which can be more irregular in shape—the coordinates of each individual node must be given, together with connectivity information that specifies how the nodes are arranged into cells. The simple example problem in this section does not have any examples of structured or unstructured grids. Conceptually, however, their use follows that exemplified above in connection with curves: a `SetGrid` method provides the information needed to set up a particular kind of grid, and an `AddVariableGroup` method attaches dependent variables to the grid. As we shall see in Section 3, in the multilevel grids we use to represent charts (coordinate patches) in a manifold, the coarsest level is a structured grid and the refined levels are unstructured grids.

### 2.2.5. Runtime

The last division of `VariableManagement` is the leaf division `Runtime` (see the diagram on the left side of Figure 2), which provides some generic functionality associated with running programs. Included are a function that returns wall time, a command that displays memory usage, and a class that handles command line options (newly accessible to programs in Fortran 2003).

The user can access all of this functionality through the object `PROGRAM_HEADER` declared in `PROGRAM_HEADER_Singleton`, which requires initialization. Our example program `SineWaveDerivative` contains the lines

```
allocate ( PROGRAM_HEADER )
call PROGRAM_HEADER % Initialize &
    ( 'SineWaveDerivative', AppendDimensionalityOption = .false. )
```

which appear at line 9 of Listing 1. The first argument to the `Initialize` method is the program name. Because this program is strictly one-dimensional, a second optional argument overrides the default behavior of appending a short string indicating the dimensionality (e.g. `'_1D'`, `'_2D'`, etc.) to the program name. Because most applications of GenASiS are expected to be written rather generically for anywhere from one to three space position space dimensions, the default behavior is to determine the dimensionality from a parameter file or command line option, or failing to find these, default to three dimensions. `PROGRAM_HEADER` has the global ('world') communicator as one of its members, and initialization of this singleton takes care of initialization of MPI (see Section 2.2.3). The `UNIT` (see “Specifiers” in Section 2.2.1) and `CONSOLE` (see Section 2.2.2) singletons are also initialized. Once `PROGRAM_HEADER` is initialized its method `ShowStatistics` can be called at any time to display total elapsed time, the amount of time spent in I/O, and current memory usage.

Another `PROGRAM_HEADER` method of significant interest is `GetParameter`. Our general approach to parameters<sup>26</sup> is to (1) set a reasonable default value in the code itself, (2) look for an overriding user-specified value in a parameter file, and (3) look for a higher-priority user-specified value among any command line options. The `PROGRAM_HEADER % GetParameter` method follows this philosophy: it informs the user of the existing (presumably default) value, and then sequentially looks for and resets that value if it is found in a parameter file and/or on the command line (with appropriate notifications along the way). The parameter file searched by default is one whose filename consists of the program name with the suffix `'_ProgramParameters'` appended; alternatively,

---

<sup>26</sup>In this context we use the term ‘parameter’ to mean a user-specified value that characterizes a problem or specifies some aspect of program execution. This usage is not quite as restrictive as the precise meaning of the Fortran `parameter` keyword; but most of the time it ends up obeying the same spirit in practice, i.e. denoting some value that remains unchanged during program execution.

a `ParametersStream` object (see “`FileSystemBasics`” in Section 2.2.4) associated with some other parameter file can be supplied as an optional argument. Our example program `SineWaveDerivative` contains the line

```
call PROGRAM_HEADER % GetParameter &
      ( SWD % Amplitude , 'Amplitude' , InputUnitOption = SineWaveUnit )
```

at line 8 of Listing 4. The final optional argument returns the units that may (optionally) be specified by the parameter input (for instance in an expression like `Amplitude=5.0~TESLA`).

### 3. MANIFOLDS

`Manifolds` is the only division of `Mathematics` that we report on in this paper. The `Mathematics` division of GenASiS contains generic mathematical constructs and solvers that are as agnostic as possible with regard to the specifics of any particular system, in order that they might in principle be used for a wide variety of physical (or other) problems. (For the place of `Mathematics` in the overall scheme of GenASiS, see Figure 1.) As illustrated in the left diagram of Figure 4, `Manifolds` is subdivided into `Charts` and `Atlases`. The middle diagram of Figure 4 shows that `Charts` includes the divisions `ChartBasics`, `Cells`, `Submeshes`, `Meshes`, `Calculus`, and `Intermeshes`. All of these are leaf divisions except `Calculus`, which includes the leaf divisions `Differences` and `Gradients` as shown in the right diagram of Figure 4.

As was done in the case of `Basics` in Section 2, we illustrate some of the functionality available through `Manifolds`—and continue to demonstrate our use of the object-oriented approach—in the context of an example problem.

#### 3.1. Another Problem and an Object-Oriented Solution

The problem we use to illustrate the functionality of `Manifolds` is somewhat similar to that discussed in Section 2. It includes the following tasks: (a) initialize a manifold with a single ‘chart’ or coordinate patch; (b) refine the chart into a series of concentric spheres with increasing resolution; (c) initialize a Gaussian function; (d) compute the gradient of this function; (e) write the output in a format suitable for visualization. We have contrived a Fortran program (Listing 10) and two supporting modules (Listings 11 and 12) to accomplish these tasks using the GenASiS `Manifolds` and `Basics` classes.

The high-level structure of this example program is very similar to that of Section 2. The drivers `SineWaveDerivative` (Listing 1) and `GaussianGradient` (Listing 10) each declare a single object and call its methods to carry out the required tasks. These objects each specialize a rather generic class to a particular functional form: `SineWaveDerivative_Form` (Listing 2) extends `WaveDerivative_Template` (Listing 3), and `GaussianGradient_Form` (Listing 11) extends `FunctionGradient_Template` (Listing 12). Beyond the members inherited from the parent class, these extensions add members parametrizing a specific functional form. The child classes also overload the `Initialize` method and override the (deferred) `Waveform` or `Function` methods that return values of the specific functional form. With the exception of `Initialize`, the methods called by the driver programs are fully declared and defined in the parent classes (`WaveDerivative_Template` and `FunctionGradient_Template`) and are agnostic with regard to the particular functional form whose derivative or gradient is to be taken. These procedures can be called in the drivers as methods of the specialized extensions (`SineDerivative_Form` and `GaussianGradient_Form`) because they are inherited from the parent class.

There are also some differences between these example programs. That the `PROGRAM_HEADER` initialization

Listing 10 program GaussianGradient

---

```
1 program GaussianGradient
2
3 use Basics
4 use GaussianGradient_Form !– See Listing 11
5
6 implicit none
7
8 type ( GaussianGradientForm ) :: GG !– See Listing 11
9
10 allocate ( PROGRAMHEADER )
11 call PROGRAMHEADER % Initialize ( 'GaussianGradient' )
12
13 call GG % Initialize ( ) !– Task (a)
14 call GG % CreateNestedSpheres ( ) !– Task (b); see Listing 17
15 call GG % SetFunction ( ) !– Task (c); see Listing 18
16 call GG % ComputeGradient ( ) !– Task (d); see Listing 16
17 call GG % Write ( ) !– Task (e); see Listing 20
18
19 deallocate ( PROGRAMHEADER )
20
21 end program GaussianGradient
```

---

Listing 11 Outline of module GaussianGradient\_Form. Used at line 4 of Listing 10.

---

```
1 module GaussianGradient_Form
2
3 use Basics
4 use FunctionGradient_Template
5 implicit none
6 private
7
8 type, public, extends ( FunctionGradientTemplate ) :: GaussianGradientForm
9   type ( MeasuredValueForm ) :: Amplitude
10   type ( MeasuredValueForm ), dimension ( 3 ) :: HalfWidth
11 contains
12   procedure, private, pass :: Initialize_GG
13   generic :: Initialize => Initialize_GG
14   procedure, private, pass :: Function
15 end type GaussianGradientForm
16
17 contains
18
19 !– Definitions of subroutine Initialize_GG and function Function omitted
20
21 end module GaussianGradient_Form
```

---

Listing 12 Outline of module `FunctionGradient_Template`. Used at line 4 of Listing 11.

---

```
1 module FunctionGradient_Template
2
3   use Basics
4   use Manifolds
5   implicit none
6   private
7
8   type, public, abstract :: FunctionGradientTemplate
9     !– Declaration of members omitted
10  contains
11    procedure, private, pass :: Initialize_FG           !– See Listing 19
12    generic, public :: Initialize => Initialize_FG
13    procedure, public, pass :: CreateNestedSpheres      !– See Listing 17
14    procedure, public, pass :: SetFunction              !– See Listing 18
15    procedure, public, pass :: ComputeGradient          !– See Listing 16
16    procedure, public, pass :: Write                    !– See Listing 20
17    procedure ( FunctionInterface ), private, pass, deferred :: Function
18  end type FunctionGradientTemplate
19
20  !– Declaration of the interface for function FunctionInterface omitted
21
22  contains
23
24  !– Definitions of subroutines Initialize_FG, CreateNestedSpheres, SetFunction,
25  ! ComputeGradient, and Write omitted
26
27 end module FunctionGradient_Template
```

---

in line 12 of Listing 10 for the `GaussianGradient` program contains only one argument (the program name), while that for the `SineWaveDerivative` program has two arguments (see Section 2.2.5), is a thread on which we can pull to unravel the distinctions. Not having the optional argument `AppendDimensionalityOption` present and set to `.false.`, a command line option allows the `GaussianGradient` program to be run in one dimension (`Dimensionality=1D`), two dimensions (`Dimensionality=2D`), or—and by default if no command line option is present—three dimensions (`Dimensionality=3D`). In contrast, the `SineWaveDerivative` program is hard coded for one dimension only. The `GaussianGradient` program can accomplish its tasks not only in anywhere from one to three dimensions, but also with mesh refinement. That it can do so with a line count comparable to that of the `SineWaveDerivative` program is only possible because it draws on functionality available through `Manifolds` classes: tasks such as setting up a grid, performing ghost cell data exchanges, and computing derivatives are easily done in `SineWaveDerivative` with `Basics` classes when only a single-level one-dimensional grid is involved, but `Manifolds` provides higher-level functionality for performing such tasks in multidimensional and multilevel contexts with comparable ease. A final difference we note is that `SineWaveDerivative` uses a periodic lattice while `GaussianGradient` has a boundary. Periodic boundary conditions are the default when using `Manifolds` classes, but a more centralized function with a fixed boundary condition better suits our purpose of demonstrating a multilevel grid of a sort suitable for gravitational collapse.

### 3.2. Illustrations of Manifolds Functionality

With this example program in mind we describe some of the functionality available in `Manifolds`. In particular we discuss in bottom-up fashion (i.e. in order of compilation) the code divisions appearing in Figure 4.

#### 3.2.1. Charts

`Charts` is the first code division under `Manifolds` in the diagram on the left side of Figure 4. The term ‘chart’ is mathematical jargon for a coordinate patch that covers an individual region of a manifold. We approximate the ideal of continuity with a finite sequence of meshes which provide, as necessary, increasing refinements of the coarsest (top-level) mesh. Our charts can be one-, two-, or three-dimensional. Its subdivisions `ChartBasics`, `Cells`, `Submeshes`, `Meshes`, `Calculus`, and `Intermeshes` are shown in the middle diagram of Figure 4. We discuss each in turn.

**ChartBasics** The first division of `Charts` is `ChartBasics`, which contains some basic definitions needed by the subsequent classes that culminate in the class `ChartForm` (see “Intermeshes” below). A number of parameters are defined in singleton objects; a few examples will appear later.

We note two classes in `ChartBasics` that are used in connection with task (b) of our example program: refine the chart into a series of concentric spheres with increasing resolution. `AdaptionForm` is an extension of `VariableGroupForm` (see “VariableGroups” in Section 2.2.1) that contains variables related to mesh refinement and coarsening. `GeometryBasicForm` is a different extension of `VariableGroupForm` that contains the most basic geometric variables associated with the cells of a mesh (edge and center positions, face areas, volume, etc.) Instances of both these classes are referenced in a subroutine called `SetRefineFlagLevel`, which marks a region for refinement; see Listing 13. (As we will see below, this routine for a single level is called at line 24 of Listing 17 for the method `CreateNestedSpheres` of `FunctionGradientTemplate`.) The argument `iLevel` (lines 1 and 5) specifies the level of the multilevel grid structure on which this routine will act. Lines 11-14 specify rank-one

Listing 13 Outline of subroutine `SetRefineFlagLevel`. Called at line 24 of Listing 17.

---

```

1  subroutine SetRefineFlagLevel ( FG, RefinementRadius, iLevel )
2
3  class ( FunctionGradientTemplate ), intent ( inout ) :: FG
4  real ( KDR ), intent ( in ) :: RefinementRadius
5  integer ( KDI ), intent ( in ) :: iLevel
6
7  integer ( KDI ) :: iV_1, iV_2  !- iValues
8
9  !- Statements omitted
10
11  associate &
12    ( X => G % Value ( iV_1 : iV_2, G % CENTER ( 1 ) ), &
13      Y => G % Value ( iV_1 : iV_2, G % CENTER ( 2 ) ), &
14      Z => G % Value ( iV_1 : iV_2, G % CENTER ( 3 ) ) )
15
16  associate ( R => sqrt ( X ** 2 + Y ** 2 + Z ** 2 ) )
17  where ( R <= RefinementRadius )
18    A % Value ( iV_1 : iV_2, A % REFINE_FLAG ) = 1.0_KDR
19  end where
20  end associate  !- R
21
22  end associate  !- X, Y, Z
23  !- Other end associate statements omitted
24
25  end subroutine SetRefineFlagLevel

```

---

array aliases `X`, `Y`, and `Z` for the coordinates of cell centers. `G` is an alias to the instance of `GeometryBasicForm` for this level. As an extension of `VariableGroupForm`, it has a data array member `Value`; the indices `iV_1` and `iV_2` specify a range of cells, and the elements of the array member `G % CENTER` index the columns of the data array containing the cell center values. Lines 16-20 specify the spherical region targeted for refinement. `R` is an alias for the distance from the cell centers to the origin. `A` is an alias to the instance of `AdaptionForm` for this level. Those cells for which `R` is less than the argument `RefinementRadius` (lines 1 and 4) have the ‘refine flag’ variable—i.e. the column of `A % Value` specified by the member `A % REFINE_FLAG`—set to a value (i.e. 1.0) that will trigger refinement.

The last class in `ChartBasics` that we mention is `ChartHeaderForm`, which contains some basic parameters and other data describing a chart. Examples include the number of dimensions, the number of cells in each dimension on the coarsest level, the minimum and maximum coordinates in each dimension on the coarsest level, the number of levels, and so on. The class `ChartForm` (see “Intermeshes” below) is an extension of `ChartHeaderForm`, so that all the members of `ChartHeaderForm` are also members of `ChartForm`.

**Cells** The second division of `Charts` is `Cells`, which contains classes that enable construction of the adaptive structure that underlies our approximate representation of a three-dimensional continuous chart. This is an oct-tree (or, in restricted use in two dimensions or even one dimension, a quad- or binary tree respectively) that enables cell-by-cell refinement. The fundamental unit of the structure is a single finite cell, which is a segment, quadrilateral, or cuboid that can be split into two, four, or eight cells in one, two, or three dimensions respectively. A cell is represented by the class `CellForm`, which contains pointers to its parent cell and to arrays of children and sibling cells. `Cells` also contains the first of several interfaces to this oct-tree, a class `CellListForm` that allows the creation of ‘cell

lists,’ or linked lists of cells. This can be used to, for example, create lists of selected parent cells in order to facilitate loops over frequently addressed subsets of cells on a particular level of the tree. Our example program does not directly reference any classes in `Cells`.

**Submeshes** The next division of `Charts` is `Submeshes`, which contains classes that define the next layer of interface to the oct-tree, a ‘submesh.’ This entity, characterized by the class `SubmeshForm`, is a grid in its own right; it consists of a subset of cells at a single level of the oct-tree, whose combined arrangement may be irregular in shape and even consist of multiple disconnected pieces. Among the members of `SubmeshForm` are two cell lists, for ‘proper’ and ‘ghost’ cells, which allow the submesh to be domain-decomposed for parallel processing in a distributed-memory environment. (Proper cells are typically the normal working computational cells assigned to a particular process. Ghost cells typically compose a partial boundary layer around the proper cells; their data is obtained through message passing with the neighboring processes that own those cells. See for instance Figure 5.) There are also members `nProperCells` and `nGhostCells` that store the number of proper and ghost cells, as well as a member `oCell` giving the offset in a data array where storage begins for the subset of cells at a particular level composing the submesh. These members are used, for example, to set the range of cells in Listing 13 discussed in “ChartBasics” above. In particular, the following lines are two of the statements omitted at line 9:

```
iV_1 = S % oCell + 1
iV_2 = S % oCell + S % nProperCells + S % nGhostCells
```

Here `S` is an alias to the instance of `SubmeshForm` comprising the body of cells interior to the boundary—either the boundary of the chart as a whole, or a coarse/fine boundary—at a particular level of refinement. With these assignments, the expression `iV_1 : iV_2` in lines 12-14 and 18 spans the range of data corresponding to all the proper and ghost cells in this submesh. A submesh and data defined on it can independently be written to the file system, most often as an unstructured mesh due to its potentially irregular shape and even multiple pieces. Examples of spherical meshes in two and three dimensions are shown in Figure 5.

**Meshes** The classes in the `Meshes` division of `Charts` combine multiple submeshes into a ‘mesh,’ characterized by the class `MeshForm`. A mesh includes four submeshes. This is illustrated in Figure 6; shown in both two and three dimensional versions are the submeshes of the Level 1 (coarsest) mesh, which is rectangular, and the submeshes of the Level 2 (first refinement) mesh, which has a spherical shape. Two of these submeshes together comprise all the cells at a particular level of the oct-tree: the ‘Interior’ submesh (or simply ‘Interior’) contains all the normal computational cells, and the ‘Exterior’ submesh includes all the cells that form a boundary layer—either the boundary of the chart as a whole, or a coarse/fine boundary at the edge of a particular level—surrounding the Interior. The other two are the ‘Parents’ and ‘Children’ submeshes, which link the cells of the Interior to the adjacent coarser and finer levels respectively. The reason for the redundancies of a Level  $i$  Parents with a Level  $i - 1$  Interior, and a Level  $i$  Children with a Level  $i + 1$  Interior, is that the Interior submeshes on each level are independently domain-decomposed: the Level  $i$  Parents and Children follow the decomposition of the Level  $i$  Interior with which they are associated, which in general may be different from the decompositions of the Interior submeshes at Levels  $i - 1$  and  $i + 1$ .

A mesh is the typical locus of data storage. The storage available in an instance of `VariableGroupForm` or its extensions (see “VariableGroups” in Section 2.2.1) is subdivided into sections reserved for data associated with the cells of the four submeshes. A given cell in the oct-tree can play any or all of three roles—‘level’ (as part of an Interior or Exterior submesh), ‘parent’ (as part of a Parents submesh), and ‘child’ (as part of a Children submesh)—and as appropriate is assigned a number for each of these roles that corresponds to a row number of the data storage array in an instance of `VariableGroupForm` or its extensions.



Listing 14 subroutine `SetFunctionLevel`. Called at line 12 of Listing 18.

---

```

1 subroutine SetFunctionLevel ( FG, iLevel )
2
3   class ( FunctionGradientTemplate ), intent ( inout ) :: FG
4   integer ( KDI ), intent ( in ) :: iLevel
5
6   !– associate statement omitted
7
8   call FG % Variable ( iLevel ) % Initialize &
9       ( [ nMaxCells, 1 ], NameOption = 'Variable', &
10        VariableOption = [ FG % FunctionName ], &
11        UnitOption = [ FG % FunctionUnit ] )
12
13   call SetFunctionSubmesh ( FG, iLevel, SUBMESH % INTERIOR )
14   if ( iLevel == 1 ) &
15       call SetFunctionSubmesh ( FG, iLevel, SUBMESH % EXTERIOR )
16
17   !– end associate statement omitted
18
19 end subroutine SetFunctionLevel

```

---

Task (c) of our example program for this section—initialize a Gaussian function—provides a further opportunity to see how this mesh-based storage is used. (We have already caught initial glimpses in Listing 13 and the discussions thereof in “ChartBasics” and “Submeshes” above.)

Consider the execution of task (c) for a single mesh at a given level of refinement in subroutine `SetFunctionLevel` in Listing 14. (As we will see below, this routine for a single level is called at line 12 of Listing 18 for the method `SetFunction` of `FunctionGradientTemplate`.) One of the members of `FunctionGradientTemplate` declared at line 9 of Listing 12 is an array of `VariableGroupForm`, each element of which stores, for a given level of refinement, the variable whose gradient is to be taken:

```
type ( VariableGroupForm ), dimension ( : ), allocatable :: Variable
```

In Listing 14, the argument `iLevel` (lines 1 and 4) specifies which level is to have the variable set. The element of `FG % Variable` for this level is initialized in lines 8-11; for more on initialization of instances of `VariableGroupForm`, see “VariableGroups” in Section 2.2.1. (The value `nMaxCells` is an alias to the maximum number of cell data values instance of `MeshForm` can have.) Separate calls to another subroutine, `SetFunctionSubmesh` shown in Listing 15, set the variable whose gradient is to be taken on the Interior submesh (line 13) and, for the coarsest level, the Exterior submesh as well (lines 14-15). Values on the Exterior of the coarsest level are set here because this constitutes the boundary of the chart as a whole; see the left panels of Figure 6. In a physics problem in which a single chart covered the entire computational domain, the Exterior submesh on the coarsest level is where physical boundary conditions would be applied; here we just apply a specified functional form whose gradient is to be taken. For the multilevel grid configuration in our example problem in this section, the Exterior submeshes of finer levels are not on the boundary on the chart as a whole; see for instance the right panels of Figure 6. In our simple example problem, values for the Exterior on the finer levels could be directly specified by the known functional form, but to illustrate what would happen in a more realistic problem, we instead set these Exterior values on finer levels later by interpolation from the coarser level (i.e. by ‘prolongation’; see “Intermeshes” below). Finally, we note that the parameters `SUBMESH % INTERIOR` and `SUBMESH % EXTERIOR` appearing in lines 13 and 15 are members a singleton `SUBMESH`; this is one of the singleton objects mentioned in “ChartBasics” above.

Listing 15 Outline of subroutine `SetFunctionSubmesh`. Called at lines 11 and 13 of Listing 14.

---

```

1 subroutine SetFunctionSubmesh ( FG, iLevel, iSubmesh )
2
3   class ( FunctionGradientTemplate ), intent ( inout ) :: FG
4   integer ( KDI ), intent ( in ) :: iLevel, iSubmesh
5
6   integer ( KDI ) :: iV_1, iV_2  !-iValues
7
8   !-associate statements omitted
9
10  iV_1 = S % oCell + 1
11  iV_2 = S % oCell + S % nProperCells
12
13  associate &
14    ( X => G % Value ( iV_1 : iV_2, G % CENTER ( 1 ) ), &
15      Y => G % Value ( iV_1 : iV_2, G % CENTER ( 2 ) ), &
16      Z => G % Value ( iV_1 : iV_2, G % CENTER ( 3 ) ), &
17      V => FG % Variable ( iLevel ) % Value ( iV_1 : iV_2, 1 ) )
18
19  V = FG % Function ( X, Y, Z )
20
21  end associate  !-X, Y, Z
22  !- Other end associate statements omitted
23
24  end subroutine SetFunctionSubmesh

```

---

We turn next to the execution of task (c) for a single submesh, shown in subroutine `SetFunctionSubmesh` in Listing 15 and called at lines 11 and 13 of Listing 14. Lines 10 and 11 are similar, but not identical, to the assignments shown in “Submeshes” above in connection with Listing 13. Here `S` is an alias to the instance of `SubmeshForm` indicated by the arguments `iLevel` and `iSubmesh` (lines 1 and 4). Unlike the case seen previously, the assignments in lines 10 and 11 cause the range `iV_1 : iV_2` in lines 14-17 to span only the proper cells in this submesh, and not the ghost cells. Despite the fact that we have a known functional form in this example problem, we do not fill in values for ghost cells here. Instead, to illustrate what is typically needed in a more realistic problem, we instead set these ghost cell values later by an ‘exchange’ of ghost cell data with neighboring processes (using message passing; see “Calculus” below). Aliases for rank-one arrays for the coordinates of the centers of the proper cells, and for the values in those cells of the variable whose gradient is to be taken, are set up in lines 13-17. The variable whose gradient is to be taken is assigned via the function call in line 19. As far as `FunctionGradientTemplate` is concerned, `FG % Function` could be anything; its definition is deferred and only an interface is specified in lines 17 and 20 of Listing 12. The functional form is specified to be a Gaussian in `GaussianGradientForm` outlined in Listing 11. This class extends `FunctionGradientTemplate` and overrides the previous deferred declaration, as indicated in lines 14 and 19 of Listing 11.

Along with being the typical locus of data storage, `MeshForm` also has members with connectivity information, that is, lists of cell numbers and their siblings’ cell numbers, in order to facilitate certain operations on particular sets of selected cells without having to walk through the oct-tree, access sibling pointers, etc. The application of boundary conditions is one example of an operation that uses this sort of connectivity information. Another is the overlapping of work and communication in time-explicit solves requiring only near-neighbor information (e.g. in hydrodynamics): ‘exchange’ cells (those whose data must be communicated to the ghost cells of other message-passing processes) can be updated, and non-blocking communication initiated, before ‘non-exchange’ cells are updated. Connectivity

Listing 16 subroutine ComputeGradient. Fits in lines 24-25 of Listing 12. Called at line 16 of Listing 10.

---

```

1  subroutine ComputeGradient ( FG )
2
3  class ( FunctionGradientTemplate ), intent ( inout ) :: FG
4
5  integer ( KDI ) :: iL  !- iLevel
6
7  !- associate statement omitted
8
9  allocate ( FG % Gradient ( C % nLevels ) )
10
11 do iL = 1, C % nLevels
12
13     associate &
14         ( M => C % Level ( iL ), &
15           V => FG % Variable ( iL ), &
16           GM => FG % Gradient ( iL ) )
17
18     call GM % Initialize ( M, V, 'Gradient_' // trim ( M % Name ) )
19     call M % StartGhostExchange ( V, SUBMESH % INTERIOR )
20     call GM % Compute ( CONNECTIVITY % NONEXCHANGE )
21     call M % FinishGhostExchange ( SUBMESH % INTERIOR )
22     call GM % Compute ( CONNECTIVITY % EXCHANGE )
23
24     end associate  !- M, V, GM
25
26 end do
27
28 !- end associate statement omitted
29
30 end subroutine ComputeGradient

```

---

information for these two categories of cells allows data to be loaded from their non-contiguous loci in variable group storage into contiguous storage in a packed variable group (see “VariableGroups” in Section 2.2.1) for efficient processing by the solver.

**Calculus** The differential structure available in a chart is realized in discrete approximation in the classes defined in the Calculus division of Charts. As shown in the right diagram of Figure 4, this division includes such operations as Differences and Gradients, which are implemented on a single level of refinement in connection with the class MeshForm just discussed above.

Task (d) of our example program for this section—compute the gradient of a function, in this case a Gaussian—utilizes the class GradientMeshForm in Gradients, and also provides an example of overlapping work and communication of ghost cell data. Note that ComputeGradient is declared as a method of FunctionGradientTemplate in line 15 of Listing 12; this subroutine is shown in Listing 16. One of the members of FunctionGradientTemplate declared at line 9 of Listing 12 is an array of GradientMeshForm, each element of which contains the gradient for a given level of refinement:

```
type ( GradientMeshForm ), dimension ( : ), allocatable :: Gradient
```

This member is allocated at line 9 of Listing 16. Here C is an alias to the instance of ChartForm used in this pro-

gram, and its member `nLevels` is one of those inherited from `ChartHeaderForm` as mentioned in “ChartBasics” above. There follows a loop over the levels of the chart in lines 11-26. Aliases `M`, `V`, and `GM` to the instances of the `MeshForm`, of the `VariableGroupForm` whose gradient is to be taken, and of the `GradientMeshForm` for a particular level of refinement are set in lines 14-16. The instance of `GradientMeshForm` is initialized in line 18; in addition to a name, it needs the instance of `MeshForm` with which it is associated, and the instance of `VariableGroupForm` whose gradient is to be taken. ‘Exchange’ cells—those on the boundary between domains assigned to different processes—need the data in their ghost cell neighbors before a gradient can be taken. This ‘non-blocking’ exchange of ghost cell data is initiated in line 19. Meanwhile the gradient values in ‘non-exchange’ cells, which do not have any ghost cell neighbors, can be computed in line 20 while the ghost cell data exchange continues in the background. Once the ghost data exchange is completed (line 21), the gradient values in exchange cells can be computed (line 22). (Connectivity lists for exchange and non-exchange cells were discussed at the end of “Meshes” above. `CONNECTIVITY` is one of the singleton objects mentioned in “ChartBasics” above; here its members `EXCHANGE` and `NONEXCHANGE` are parameters identifying the corresponding connectivity lists.)

**Intermeshes** The classes in the `Intermeshes` division of `Charts` bring together the structures described above into a ‘chart,’ or coordinate patch, characterized by the class `ChartForm`. As already suggested above, `ChartForm` has among its members an array of `MeshForm`, each element of which corresponds to a (potential) level of the refinable oct-tree. This member of `ChartForm`, an array of `MeshForm`, is named `Level`. Initialization of an object of class `ChartForm` automatically includes setting up the coarsest level of the chart in accordance with parameters mentioned above in “ChartBasics.”

Each successive level of the chart is a refinement of the previous level, allowing the chart to approximate the ideal of continuity as needed. An example, generated by our sample problem for this section, is shown in Figure 7. It consists of 10 levels and illustrates the dynamic range in length scales accessed during the gravitational collapse of the core of a massive star. The levels of this chart are generated by the method `CreateNestedSpheres` of `FunctionGradientTemplate`, declared in line 13 of Listing 12; this subroutine is shown in Listing 17. The local variable `nLevels` is given a default value of 1 (lines 5, 10). A larger value of `nLevels` can be specified in a parameter file and/or the command line using the `GetParameter` method of the `PROGRAM.HEADER` singleton (line 11; see Section 2.2.5). If `nLevels > 1` (line 14), the radii of spherical meshes at levels  $2 \leq iL \leq nLevels$  are computed (lines 16-21), where the local variable `iL` (line 5) indexes the levels. The refinements that set up levels 2 through `nLevels` occur in lines 23-27. The routine `SetRefineFlagLevel` called in line 24 was given in Listing 13 and discussed in “ChartBasics” earlier in this section. Invocations of the methods `AddLevel` and `SetLevel` of `ChartForm` (lines 25-26) prepare a new, refined level. `C` is an alias to the instance of `ChartForm` used in this program. The method `AddLevel` performs the initializations and storage allocations needed for a new level, and increments the member `C % nLevels`. The refinement actually takes place in the call to `C % SetLevel`, along with the setting of the geometry, boundary, and connectivity of the new level.

The interpolation of Level  $i$  data to Level  $i + 1$  is called ‘prolongation,’ and the averaging of Level  $i$  data to Level  $i - 1$  is called ‘restriction.’ These operations are illustrated in Figure 8. The first step of prolongation (left panel in Figure 8) is to interpolate data from the Level  $i$  Interior submesh to the Level  $i$  Children submesh.<sup>27</sup> The Level  $i$  Children submesh is refined relative to the Level  $i$  Interior with which it is associated. But in a distributed-memory message passing environment, the Level  $i$  Children follows the domain decomposition of the Level  $i$  Interior, as mentioned in “Meshes” above, making the interpolation step a local operation. This is why both the Interior and Children submeshes are part of the same Level  $i$  mesh. But the distribution of the Level  $i + 1$  Interior cells

---

<sup>27</sup>Gradients used in the interpolation optionally can be computed with a slope limiter in order to respect discontinuities.

Listing 17 subroutine `CreateNestedSpheres`. Fits in lines 24-25 of Listing 12. Called at line 14 of Listing 10.

---

```
1 subroutine CreateNestedSpheres ( FG )
2
3   class ( FunctionGradientTemplate ), intent ( inout ) :: FG
4
5   integer ( KDI ) :: iL, nLevels
6   real ( KDR ), dimension ( : ), allocatable :: MeshRadius
7
8   !– associate statement omitted
9
10  nLevels = 1
11  call PROGRAMHEADER % GetParameter &
12      ( nLevels, 'nLevels', IgnorabilityOption = CONSOLE % INFO_1 )
13
14  if ( nLevels == 1 ) return
15
16  allocate ( MeshRadius ( 2 : nLevels ) )
17  MeshRadius ( 2 ) &
18      = 0.5_KDR * minval ( C % MaxCoordinate ( 1 : C % nDimensions ) )
19  do iL = 3, nLevels
20      MeshRadius ( iL ) = 0.5_KDR * MeshRadius ( iL – 1 )
21  end do
22
23  do iL = 1, nLevels – 1
24      call SetRefineFlagLevel ( FG, MeshRadius ( iL + 1 ), iL ) !– See Listing 13
25      call C % AddLevel ( )
26      call C % SetLevel ( IsNewLevel = .true., iLevel = C % nLevels )
27  end do
28
29  !– end associate statement omitted
30
31 end subroutine CreateNestedSpheres
```

---

Listing 18 subroutine `SetFunction`. Fits in lines 24-25 of Listing 12. Called at line 15 of Listing 10.

---

```

1  subroutine SetFunction ( FG )
2
3  class ( FunctionGradientTemplate ), intent ( inout ) :: FG
4
5  integer ( KDI ) :: iL  !- iLevel
6
7  !- associate statement omitted
8
9  allocate ( FG % Variable ( C % nLevels ) )
10
11 do iL = 1, C % nLevels
12     call SetFunctionLevel ( FG, iL )  !- See Listing 14
13 end do
14
15 do iL = C % nLevels, 2, -1
16     call C % Restrict &
17         ( FG % Variable ( iL ), FG % Variable ( iL - 1 ), &
18         Interior = .true., Exterior = .false., iLevel = iL )
19 end do
20
21 do iL = 1, C % nLevels - 1
22     call C % Prolong &
23         ( FG % Variable ( iL ), FG % Variable ( iL + 1 ), &
24         Interior = .false., Exterior = .true., iLevel = iL )
25 end do
26
27 !- end associate statement omitted
28
29 end subroutine SetFunction

```

---

among processes in general may differ from that of the Level  $i$  Interior, as the Interior submeshes on each level are independently domain-decomposed. Thus the second and final step of prolongation is communication, in general involving message passing, between from the Level  $i$  Children to the Level  $i + 1$  Interior and/or Exterior. Similarly, restriction (right panel in Figure 8) involves first an averaging of Level  $i$  Interior and/or Exterior data to the Level  $i$  Parents (a local operation), followed by communication from the Level  $i$  Parents to the Level  $i - 1$  Interior (which in general involves message passing).

In our example program, prolongation and restriction are performed in the `SetFunction` method of `FunctionGradientTemplate`. This method is declared in line 14 of Listing 12 and the subroutine is shown in Listing 18. The member `Variable` of `FunctionGradientTemplate`, allocated in line 9, is an array of `VariableGroupForm` that we encountered in “Meshes” above. Each element of the array stores, for a given level of refinement, the variable whose gradient is to be taken. As before, `C` is an alias to the instance of `ChartForm` used in this program. The routine `SetFunctionLevel`, given in Listing 14 and discussed in “Meshes” above, is called for all levels in lines 11-13. An example of restriction from the finest to the coarsest level follows in lines 15-19. In this toy problem the example is gratuitous because the Interior at each level already has been initialized from a known analytic function. In a more realistic problem, however, restriction of more accurate data from a finer level to a coarser one can be a necessary operation. In this particular case the arguments specify that only the Interior data on Level  $i$  is to be averaged to Level  $i - 1$ ; see the right panel of Figure 8. Finally, lines 21-25 illustrate prolongation from the coarsest to the finest level. In this case data is interpolated from the Interior of Level  $i$  to the Exterior of

Level  $i + 1$ , providing boundary data needed in this example to compute the gradient at the finer levels. See the left panel of Figure 8.

### 3.2.2. Atlases

`Atlases` is the second and final code division under `Manifolds` in the diagram on the left side of Figure 4. The term ‘atlas’ is mathematical parlance for a collection of charts that covers a manifold. A sufficiently simple manifold can be described with a single chart; this is the case with the `GaussianGradient` example problem in this section. But one can imagine many reasons to use multiple charts. For example, a ‘yin-yang’ atlas includes two overlapping charts that—like the two pieces of leather that form the surface of a baseball—cover a three-dimensional space with separate spherical coordinates in such a way as to avoid coordinate singularities at the origin and on the polar axis (e.g. [Wongwathanarat et al. 2010](#)). One could efficiently handle both spherically symmetric radial gravitational collapse and multidimensional phenomena at smaller radius by marrying a central three-dimensional Cartesian mesh to a one-dimensional radial mesh that begins near the surface of the Cartesian box and extends to much larger radius ([Scheidegger et al. 2008](#)). Choices of algorithm and approximation might suggest that different pieces of physics be treated on separate charts, with some facility for interpolation between them ([Rampp and Janka 2002](#); [Buras et al. 2006](#)). A binary stellar system could be handled by covering the two bodies with separate meshes that move within a larger mesh to follow the orbital motion ([Scheel et al. 2006](#)). Some manifolds, such as some spaces described by general relativity, may be sufficiently complicated as to require multiple charts for a reasonable description. Another example is the phase space—position space plus momentum space—needed for relativistic kinetic theory, which in mathematical idealization contains an infinite number of charts: this is a ‘tangent bundle,’ consisting of a (curved, in the general case) base space, i.e. position space, together with a flat tangent space, i.e. a momentum space, at every point of the base space.

The class `AtlasForm` has among its members an array `Chart` of `ChartForm`. In our example program, an instance of `AtlasForm` is one of the members of `FunctionGradientTemplate` declared at line 9 of Listing 12:

```
type ( AtlasForm ), allocatable :: Atlas
```

This member is initialized in the method `Initialize_FG` of `FunctionGradientTemplate`, declared in line 11 of Listing 12 and shown in Listing 19. Allocation of `FG % Atlas` occurs in line 10 and an alias `A` to it is specified in line 11. Initialization of the atlas is accomplished with the call in line 13. After initialization one or more charts must be added. This example involves only a single chart, which is added with a call to the method `AddChart` in line 14. The name of the chart, provided as an argument in this call, is used to find the parameter file needed to set up the chart. In previous listings in this section we have used an alias `C` to refer to the single chart used in our example program. This alias is set with the expression

```
associate ( C => FG % Atlas % Chart ( 1 ) )
```

which would appear for instance at line 7 of Listing 16, line 8 of Listing 17, and line 7 of Listing 18.

`AtlasForm` has methods for writing its charts, and data thereon, to output files. These are called in the method `Write` of `FunctionGradientTemplate`, declared in line 16 of Listing 12 and shown in Listing 20. Aliases `A` and `C` for the atlas and its chart are set in line 8. An instance of `GridImageStreamForm` (lines 6, 19-20, 33; see “GridImageBasics” in Section 2.2.4) provides access to an output file. The writing of the atlas to this output file is prepared by a call to the method `OpenStream` of `AtlasForm` (line 21), and completed by a call to the method `CloseStream` (line 32). The fields (i.e. variables) to be written are specified by calls to the method `AddFieldImage`. Three of these, in lines 23-25, are straightforward: `C % Geometry`, `C % Adaption`, and `FG`



Listing 19 subroutine `Initialize_FG`. Fits in lines 24-25 of Listing 12.

---

```

1  subroutine Initialize_FG ( FG, FunctionName, FunctionUnit )
2
3      class ( FunctionGradientTemplate ), intent ( inout ) :: FG
4      character ( * ), intent ( in ) :: FunctionName
5      type ( UnitForm ), intent ( in ) :: FunctionUnit
6
7      FG % FunctionUnit = FunctionUnit
8      FG % FunctionName = trim ( FunctionName )
9
10     allocate ( FG % Atlas )
11     associate ( A => FG % Atlas )
12
13     call A % Initialize ( PROGRAM_HEADER % Communicator, PROGRAM_HEADER % Name )
14     call A % AddChart ( A % Name )
15
16     end associate !-A
17
18 end subroutine Initialize_FG

```

---

% Variable (see “ChartBasics” and “Meshes” in Section 3.2.1) are already arrays of `VariableGroupForm` over the levels of the chart corresponding to the argument `iChart = 1`. Output of the gradient is a bit more complicated because the storage is not innately arranged as arrays over levels as expected by the `AddFieldImage` method. A rank-two array `GradientOutput` of `VariableGroupForm`, whose first index spans number of levels and whose second index spans the number of dimensions, is prepared in lines 10-17 and used in calls to `AddFieldImage` in lines 26-28. The write itself occurs with the call to the `Write` method of `AtlasForm` in line 30.

Plots made from data produced by our `GaussianGradient` example program with `nLevels=10` are presented in Figures 9-12. These show Gaussian functions in two and three dimensions and the  $x$  component of their gradients, illustrating the range of scales encountered in the gravitational collapse of the core of a massive star and the continuity of the Gaussian function and its gradient at boundaries between adjacent levels of refinement.

## 4. CONCLUSION

This paper is the first in a series on GenASiS (*General Astrophysical Simulation System*), a new code ultimately aimed at state-of-the-art simulations of core-collapse supernovae and other astrophysical problems. Distinguishing features include an object-oriented approach and cell-by-cell mesh refinement. The core of GenASiS includes three major divisions: `Basics`, which contains some basic utilitarian functionality for large-scale simulations on distributed-memory supercomputers; `Mathematics`, which includes generic mathematical constructs and solvers that are as agnostic as possible with regard to the specifics of any particular system; and `Physics`, which sets up physical spaces associated with various theories of spacetime (including gravity), defines various forms of stress-energy, and combines these into ‘universes.’ As illustrated in Figure 1, these three divisions are present in each of the two highest-level divisions shown: `Modules`, which contains class implementations; and `Programs`, which contains a unit test for each class. `Programs` also has a division `Applications` containing drivers for applications beyond unit tests. In this first paper we focus on `Basics` and on the `Manifolds` division of `Mathematics`. These contain fundamental capabilities that underpin implementations of the solvers and physics needed for the simulation of core-collapse supernovae and other astrophysical systems.

Listing 20 subroutine `Write`. Fits in lines 24-25 of Listing 12. Called at line 17 of Listing 10.

---

```
1  subroutine Write ( FG )
2
3      class ( FunctionGradientTemplate ), intent ( inout ) :: FG
4
5      integer ( KDI ) :: iL, iD
6      type ( GridImageStreamForm ) :: GIS
7
8      associate ( A => FG % Atlas , C => FG % Atlas % Chart ( 1 ) )
9
10     allocate ( FG % GradientOutput ( C % nLevels , C % nDimensions ) )
11     do iL = 1, C % nLevels
12         associate ( GM => FG % Gradient ( iL ), GO => FG % GradientOutput ( iL , : ) )
13         do iD = 1, C % nDimensions
14             call GO ( iD ) % Initialize ( GM % Output ( iD ) )
15         end do
16     end associate !-G, GO
17 end do
18
19     call GIS % Initialize ( C % Name, CommunicatorOption = C % Communicator )
20     call GIS % Open ( GIS % ACCESS_CREATE )
21     call A % OpenStream ( GIS )
22
23     call A % AddFieldImage ( C % Geometry , iChart = 1 )
24     call A % AddFieldImage ( C % Adaption , iChart = 1 )
25     call A % AddFieldImage ( FG % Variable , iChart = 1 )
26     do iD = 1, C % nDimensions
27         call A % AddFieldImage ( FG % GradientOutput ( :, iD ), iChart = 1 )
28     end do
29
30     call A % Write ( )
31
32     call A % CloseStream ( )
33     call GIS % Close ( )
34
35     end associate !-A, C
36
37 end subroutine Write
```

---

The content of `Basics` is outlined in Figure 2. The first division, `VariableManagement`, contains several subdivisions shown in the right diagram of Figure 2. `Specifiers` contains classes used in the specification of number and character variables, as well as mathematical and physical constants and a means of dealing with units. `ArrayOperations` includes some basic operations on arrays. `ArrayArrays` contains classes that can be used to form arrays of arrays; one application is the construction of so-called ragged arrays. `VariableGroups` contains classes we use extensively in handling collections of variables, especially sets of related physical fields. After `VariableManagement` in the left diagram of Figure 2 comes `Display`, which contains infrastructure for displaying messages and variables to the standard output in a uniform and orderly way. The classes in `MessagePassing`, found in its subdivisions `MessagePassingBasics`, `PointToPoint`, and `Collective` (see the middle right diagram in Figure 2), abstract the data and methods useful for working in a message passing parallel computing environment. `FileSystem` is next in the left diagram of Figure 2; the classes in `FileSystemBasics`, `GridImageBasics`, `CurveImages`, `StructuredGridImages`, and `UnstructuredGridImages` (see the middle left diagram in Figure 2) handle I/O to disk, in part by providing a façade facilitating interaction with sophisticated I/O libraries. Finally, `Runtime` provides a `PROGRAM.HEADER` object giving access to such facilities such as a function that returns wall time, a command that displays memory usage, and a class that handles command line options (newly accessible to programs in Fortran 2003).

The content of `Manifolds`, a division of `Mathematics`, is outlined in Figure 4. The first division in the left diagram of Figure 4, `Charts`, contains several subdivisions shown in the middle diagram of Figure 4; these comprise the infrastructure representing a coordinate patch that covers an individual region of a manifold. We approximate the ideal of continuity with a finite sequence of meshes which provide, as necessary, increasing refinements of the coarsest (top-level) mesh. Our charts can be one-, two-, or three-dimensional. `ChartBasics` contains some basic definitions needed by the subsequent classes that culminate in the class `ChartForm`. `Cells` contains classes that enable construction of the adaptive structure that underlies our approximate representation of a three-dimensional continuous chart—an oct-tree (or, in restricted use in two dimensions or even one dimension, a quad- or binary tree respectively) that enables cell-by-cell refinement. `Submeshes` includes classes that culminate in `SubmeshForm`, which embodies a grid consisting of a subset of cells at a single level of the oct-tree, whose combined arrangement may be irregular in shape and even consist of multiple disconnected pieces. The class `MeshForm` in `Meshes` integrates four submeshes into a single level of the multilevel chart; the `Interior` and `Exterior` submeshes comprise all the cells at a particular level of the oct-tree, and the `Parents` and `Children` submeshes link the cells of the `Interior` to the adjacent coarser and finer levels respectively (see Figure 6). The differential structure available in a chart is realized in discrete approximation in the classes defined in `Calculus`; as shown in the right diagram of Figure 4, this includes the operations `Differences` and `Gradients`, which are implemented on a single level of refinement in connection with the class `MeshForm`. The classes in `Intermeshes` culminate in `ChartForm`, whose array of `MeshForm` allows for a refinable chart or coordinate patch, with capabilities for interpolation of data to a more refined level (prolongation) and averaging of data to coarser level (restriction). After `Charts`, the second and final division in `Manifolds` in the left diagram of Figure 4 is `Atlases`, which defines a collection of charts covering a manifold.

We present two sample problems, one each for `Basics` and `Manifolds`. The `SineWaveDerivative` program, which illustrates the divisions of `Basics`, computes the derivative of a sine wave in one dimension on a periodic lattice, with output shown in Figure 3. The `GaussianGradient` program, which illustrates the divisions of `Manifolds`, computes the gradient of a Gaussian function in any of one to three dimensions on a multilevel grid of a sort suitable for gravitational collapse, with output shown in Figures 9-12. That it can do so with a line count comparable to that of the `SineWaveDerivative` program is only possible because it draws on functionality available through `Manifolds` classes: tasks such as setting up a grid, performing ghost cell data exchanges, and computing derivatives are easily done in `SineWaveDerivative` with `Basics` classes when only a single-level one-dimensional grid is involved, but `Manifolds` provides higher-level functionality for performing such tasks in

multidimensional and multilevel contexts with comparable ease.

Listings drawn from these sample programs serve not only to exhibit the contents of the `Basics` and `Manifolds` divisions of GenASiS, but also to illustrate our use of object-oriented design principles using the features of Fortran 2003 that support this programming paradigm (e.g. Reid 2007; Adams et al. 2008). These principles promote code reusability by facilitating generality, extensibility, and maintainability. In our context, an object-oriented approach enables the flexibility connoted by the ‘General’ in GenASiS—the capacity of the code to include and refer to multiple algorithms, solvers, and physics and numerics choices with the same abstracted names and/or interfaces. *Abstraction* identifies the major concepts required by a program, without specifying the details of implementation. *Encapsulation* bundles together and controls access to the data (or *members*) and actions (or *methods*) associated with a particular concept into a self-contained unit—a *class*. Done well, abstraction and encapsulation lead to *decoupling*—the separation of code into building blocks that are as independent and reusable as possible. Reusability is further enhanced by *polymorphism*, which enables multiple versions or implementations of the same basic concept to be referred to and used interchangeably. Closely related is *inheritance*, which allows new (*child*) classes to be formed from prior (*parent*) classes: existing members and methods are retained or modified, and new members and methods can be added.

By way of applying the principles of abstraction, encapsulation, and decoupling, we have established conventions for implementing classes via Fortran `modules` and derived `types`. These conventions are illustrated in Listings 2 and 3 from the `SineWaveDerivative` example, and Listings 11 and 12 from the `GaussianGradient` example. A single `module` containing a single derived `type` definition comprises a class. The names of the module and the derived type correspond to each other (up to an extra underscore preceding the suffix in the module name). Derived types with data components have been available since Fortran 90; these data components are the *members* of the class. But derived types with procedure components are new to Fortran 2003; these procedure components are the *methods* of the class. The language allows for two kinds of procedure components: procedure pointer variables, and type-bound procedures. We generally use the latter. Type-bound procedures are declared in the `contains` section of the derived type definition, while the subroutine or functions themselves are placed in the `contains` section of the module.

Features new to Fortran 2003 also allow us to implement the principles of inheritance and polymorphism. The new `extends` keyword allows a new derived type (a child) to directly inherit the members and methods of a previously defined derived type (the parent), so that these components can be referenced directly as if they had been declared in the new child class itself. The new derived type can add additional members and methods to those already available through the parent type. The child also can modify methods of the parent in two ways. First, with the new `generic` keyword it can *overload* a type-bound procedure name that is resolved to one of multiple procedures depending on the argument list. Second, it can *override* a type-bound procedure with a new subroutine or function exhibiting different behavior than the one in the parent class that it replaces. We also note the new capability of defining a derived type as `abstract`, and that includes deferred methods for which only an interface—and not the subroutine or function itself—are provided. Objects of an abstract type cannot be instantiated (i.e. variables of an abstract type cannot be declared); only objects of an extension of the abstract type, with type-bound procedures that override the deferred methods, can be instantiated.

These mechanisms—which prove so useful in developing solvers and physics capabilities for GenASiS—are illustrated in simple cases in Listings 2 and 3 from the `SineWaveDerivative` example, and Listings 11 and 12 from the `GaussianGradient` example. In each of these problems the derivative or gradient of only a single function is taken. But in each pair of listings, the parent class is an abstract template that, while containing some 80% of the lines of code, does not depend upon the particular functional form at all. The child class in each pair of listings serves only to specify a particular functional form. It is evident that it would be simple and easy to make several other extensions for other functional forms. Of course, in these trivial examples, less sophisticated techniques (such

as passing the name of a function as a subroutine argument) could achieve the same purpose; but in more complex and realistic situations, trying to achieve similar generality in Fortran with pre-Fortran 2003 language features fast becomes difficult and unwieldy. Language constructs enabling object-oriented programming provide a much better means of achieving the same sort of code reusability. The point is that the principles of inheritance and polymorphism—exemplified here in the mechanisms of `type` extension and method overriding—make it much easier to allow lower-level code to access higher-level code. This language functionality greatly facilitates, for instance, the separation of the `Mathematics` and `Physics` divisions of GenASiS. For example, solvers for generic classes of equations can be written in `Mathematics`, and then invoked later by a range of different systems whose details are specified in `Physics`. This tremendously enhances the ease and transparency with which one can develop (for example) versatile and widely-applicable solvers.

An object-oriented approach need not ruin code performance. This is because the floating-point operations on which the bulk of computational effort is (hopefully) spent can be isolated in low-level kernels expressed in terms of elementary loops over operations on array variables of intrinsic data types, which the compiler can handle well. Consider for example the main computational task in the `SineWaveDerivative` example: computing the derivative of the wave. The wave and derivative data are stored in a member of an object of `VariableGroupForm`, in such a way (see “VariableGroups” in Section 2.2.1) that reference to that data may at first look intimidating—not only to humans, but perhaps also to compilers! However the actual operations on that data look straightforward to both of these audiences:

```
do i = WD % iFirst + 1, WD % iLast - 1
  dfdx ( i ) = ( f ( i + 1 ) - f ( i - 1 ) ) / ( 2.0 * dx )
end do
```

Here `f` and `dfdx` are simply rank-one `real` dummy arguments of a kernel subroutine. The actual arguments are columns of the variable group’s rank-two data array (and are therefore contiguous in memory), but reference to them here is plenty simple enough for the compiler to be able to apply any optimizations that may be useful. Once again, this example problem is trivial, this time in the sense that there is so little data on which to work; but this same principle, namely of isolating computational work in kernel subroutines with basic loops over operations on simple dummy arguments of intrinsic data types, continues to be useful when the amount of data is much greater. Our computational kernels throughout GenASiS—including in the `Calculus` classes that compute differences and gradients for the `GaussianGradient` example—take this approach.

Subsequent papers in this series will document additions to `Mathematics` and `Physics` that will make GenASiS suitable for multiphysics simulations of core-collapse supernovae and other astrophysical problems. Development of additional solvers and physics are already underway. Capabilities for nonrelativistic hydrodynamics will be reported first (Cardall et al. 2012). We have implemented magnetohydrodynamics (MHD) capabilities in a ‘draft version’ of GenASiS (Endeve et al. 2012), which has been used to study standing accretion shock instability (SASI)-driven magnetic field amplification (Endeve et al. 2010, 2012). In the current version of GenASiS we plan to implement MHD capabilities based on the HLLD solver (Miyoshi and Kusano 2005; Mignone et al. 2009). For Newtonian gravity, a multilevel Poisson solver will use a distributed FFT solver (Budiardja and Cardall 2011) for the coarsest level, in conjunction with finite-difference solves on individual levels, in a multigrid approach. Work on general relativistic gravity—a major undertaking—is also underway in GenASiS (Tsatsin et al. 2011). The greatest challenge of all is neutrino transport. Building on our group’s past experience (Liebendörfer et al. 2004; Bruenn et al. 2009), theoretical developments (Cardall and Mezzacappa 2003; Cardall et al. 2005), and initial forays into transport in the earliest version of GenASiS (Cardall et al. 2006; Cardall 2009), we plan to deploy a multigrid approach here as well, using a multigroup Variable Eddington Tensor formulation (Endeve et al. 2012, in preparation) with closures of increasing sophistication, ultimately culminating in a full ‘Boltzmann solver.’

This research was supported by the Office of Advanced Scientific Computing Research and the Office of Nuclear Physics, U.S. Department of Energy. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory provided through the INCITE program.

## REFERENCES

- Adams, J. C., Brainerd, W. S., Hendrickson, R. A., Maine, R. E., Martin, J. T., and Smith, B. T. (2008). *The Fortran 2003 Handbook: The Complete Syntax, Features and Procedures*. Springer. [3](#), [40](#)
- Agertz, O., Moore, B., Stadel, J., Potter, D., Miniati, F., Read, J., Mayer, L., Gawryszczak, A., Kravtsov, A., Nordlund, A. k., Pearce, F., Quilis, V., Rudd, D., Springel, V., Stone, J., Tasker, E., Teyssier, R., Wadsley, J., and Walder, R. (2007). Fundamental differences between SPH and grid methods. *Monthly Notices of the Royal Astronomical Society*, 380(3):963–978. [3](#)
- Almgren, A. S., Beckner, V. E., Bell, J. B., Day, M. S., Howell, L. H., Joggerst, C. C., Lijewski, M. J., Nonaka, A., Singer, M., and Zingale, M. (2010). CASTRO: A NEW COMPRESSIBLE ASTROPHYSICAL SOLVER. I. HYDRODYNAMICS AND SELF-GRAVITY. *The Astrophysical Journal*, 715(2):1221–1238. [4](#)
- Berger, M. and Collela, P. (1989). Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82(1):64–84. [4](#)
- Berger, M. and Oliger, J. (1984). Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53(3):484–512. [4](#)
- Bruenn, S. W., Mezzacappa, A., Hix, W. R., Blondin, J. M., Marronetti, P., Messer, O. E. B., Dirk, C. J., and Yoshida, S. (2009). 2D and 3D core-collapse supernovae simulation results obtained with the CHIMERA code. *Journal of Physics: Conference Series*, 180:012018. [2](#), [4](#), [41](#)
- Budiardja, R. D. and Cardall, C. Y. (2011). Parallel FFT-based Poisson solver for isolated three-dimensional systems. *Computer Physics Communications*, 182(10):2265–2275. [41](#)
- Buras, R., Rampp, M., Janka, H.-T., and Kifonidis, K. (2006). Two-dimensional hydrodynamic core-collapse supernova simulations with spectral neutrino transport. *Astronomy and Astrophysics*, 447(3):1049–1092. [4](#), [36](#)
- Cardall, C., Budiardja, R., Endeve, E., and Mezzacappa, A. (2012). GenASiS: General Astrophysical Simulation System. II. Nonrelativistic Hydrodynamics. *ApJS*, *submitted*. [41](#)
- Cardall, C., Lentz, E., and Mezzacappa, A. (2005). Conservative special relativistic radiative transfer for multidimensional astrophysical simulations: Motivation and elaboration. *Physical Review D*, 72(4). [41](#)
- Cardall, C. and Mezzacappa, A. (2003). Conservative formulations of general relativistic kinetic theory. *Physical Review D*, 68(2). [41](#)
- Cardall, C. Y. (2009). An approach to neutrino radiative transfer in supernova simulations. In Kanschat, G., Meinköhn, E., Rannacher, R., and Wehrse, R., editors, *Numerical Methods in Multidimensional Radiative Transfer*, pages 27–37, Berlin Heidelberg. Springer. [41](#)
- Cardall, C. Y., Razoumov, A. O., Endeve, E., and Mezzacappa, A. (2006). The Long Term: Six-dimensional Core-collapse Supernova Models. In Mezzacappa, A. and Fuller, G. M., editors, *Open Issues in Core Collapse Supernova Theory*. World Scientific Publishing Company, London, England. [41](#)



- Endeve, E., Cardall, C. Y., Budiardja, R. D., Beck, S. W., Bejnood, A., Toedte, R. R., Mezzacappa, A., and Blondin, John, M. (2012). Turbulent Magnetic Field Amplification from Spiral SASI Modes: Implications for Core-Collapse Supernovae and Proto-Neutron Star Magnetization. *The Astrophysical Journal*, 751(26). [41](#)
- Endeve, E., Cardall, C. Y., Budiardja, R. D., and Mezzacappa, A. (2010). Generation of Magnetic Fields by The Stationary Accretion Shock Instability. *The Astrophysical Journal*, 713(2):1219–1243. [41](#)
- Endeve, E., Y Cardall, C., Budiardja, R., and Mezzacappa, A. (2012). Turbulent magnetic field amplification from spiral SASI modes in core-collapse supernovae. *Journal of Physics: Conference Series*, in press (*arXiv:1203.3385*). [41](#)
- Fryer, C. L., Rockefeller, G., and Warren, M. S. (2006). SNSPH: A Parallel Three-dimensional Smoothed Particle Radiation Hydrodynamics Code. *ApJ*, 643:292–305. [4](#)
- Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional. [18](#)
- Gittings, M., Weaver, R., Clover, M., Betlach, T., Byrne, N., Coker, R., Dendy, E., Hueckstaedt, R., New, K., Oakes, W. R., Ranta, D., and Stefan, R. (2008). The RAGE radiation-hydrodynamic code. *Computational Science & Discovery*, 1(1):015005. [4](#)
- Gropp, W., Lusk, E., and Skjellum, A. (1999). *Using MPI - 2nd Edition: Portable Parallel Programming with the Message Passing Interface (Scientific and Engineering Computation)*. The MIT Press, 2 edition. [18](#)
- Janka, H., Langanke, K., Marek, A., Martinezpinedo, G., and Muller, B. (2007). Theory of core-collapse supernovae. *Physics Reports*, 442(1-6):38–74. [2](#)
- Janka, H.-T. (2012). Explosion Mechanisms of Core-Collapse Supernovae. *ArXiv:1206.2503*. [2](#)
- Khokhlov, A. (1998). Fully Threaded Tree Algorithms for Adaptive Refinement Fluid Dynamics Simulations. *Journal of Computational Physics*, 143(2):519–543. [4](#)
- Kotake, K., Sato, K., and Takahashi, K. (2006). Explosion mechanism, neutrino burst and gravitational wave in core-collapse supernovae. *Reports on Progress in Physics*, 69:971–1143. [2](#)
- Liebendörfer, M., Messer, O. E. B., Mezzacappa, A., Bruenn, S. W., Cardall, C. Y., and Thielemann, F.-K. (2004). A Finite Difference Representation of Neutrino Radiation Hydrodynamics in Spherically Symmetric General Relativistic Spacetime. *ApJS*, 150:263–316. [4](#), [41](#)
- Livne, E., Dessart, L., Burrows, A., and Meakin, C. a. (2007). A Twodimensional Magnetohydrodynamics Scheme for General Unstructured Grids. *The Astrophysical Journal Supplement Series*, 170(1):187–202. [4](#)
- MacNeice, P. (2000). PARAMESH: A parallel adaptive mesh refinement community toolkit. *Computer Physics Communications*, 126(3):330–354. [4](#)
- Mezzacappa, A. (2005). ASCERTAINING THE CORE COLLAPSE SUPERNOVA MECHANISM: The State of the Art and the Road Ahead. *Annual Review of Nuclear and Particle Science*, 55(1):467–515. [2](#)
- Mignone, a., Ugliano, M., and Bodo, G. (2009). A five-wave Harten-Lax-van Leer Riemann solver for relativistic magnetohydrodynamics. *Monthly Notices of the Royal Astronomical Society*, 393(4):1141–1156. [41](#)
- Miyoshi, T. and Kusano, K. (2005). A multi-state HLL approximate Riemann solver for ideal magnetohydrodynamics. *Journal of Computational Physics*, 208(1):315–344. [41](#)



- O’Shea, B. W., Bryan, G., Bordner, J., Norman, M. L., Abel, T., Harkness, R., and Kritsuk, A. (2005). Introducing enzo, an amr cosmology application. In Plewa, T., Linde, T., and Weirs, V. G., editors, *Adaptive Mesh Refinement - Theory and Applications*, volume 41 of *Lecture Notes in Computational Science and Engineering*, pages 341–350, Berlin. Springer. [4](#)
- Price, D. J. (2008). Modelling discontinuities and Kelvin Helmholtz instabilities in SPH. *Journal of Computational Physics*, 227:10040–10057. [4](#)
- Rampp, M. and Janka, H.-T. (2002). Radiation hydrodynamics with neutrinos. Variable Eddington factor method for core-collapse supernova simulations. *A&A*, 396:361–392. [4](#), [36](#)
- Reid, J. (2007). The new features of fortran 2003. *SIGPLAN Fortran Forum*, 26:10–33. [3](#), [7](#), [40](#)
- Ricker, P. M. (2008). A Direct Multigrid Poisson Solver for OctTree Adaptive Meshes. *The Astrophysical Journal Supplement Series*, 176(1):293–300. [4](#)
- Rijkhorst, E.-J., Plewa, T., Dubey, A., and Mellema, G. (2006). Hybrid characteristics: 3D radiative transfer for parallel adaptive mesh refinement hydrodynamics. *A&A*, 452:907–920. [4](#)
- Scheel, M. A., Pfeiffer, H. P., Lindblom, L., Kidder, L. E., Rinne, O., and Teukolsky, S. A. (2006). Solving Einstein’s equations with dual coordinate frames. *Phys. Rev. D*, 74(10):104006–+. [4](#), [36](#)
- Scheidegger, S., Fischer, T., Whitehouse, S. C., and Liebendörfer, M. (2008). Gravitational waves from 3D MHD core collapse simulations. *A&A*, 490:231–241. [4](#), [36](#)
- Springel, V. (2010a). E pur si muove: Galilean-invariant cosmological hydrodynamical simulations on a moving mesh. *MNRAS*, 401:791–851. [4](#)
- Springel, V. (2010b). Smoothed Particle Hydrodynamics in Astrophysics. *ARA&A*, 48:391–430. [4](#)
- Sumiyoshi, K., Yamada, S., Suzuki, H., Shen, H., Chiba, S., and Toki, H. (2005). Postbounce Evolution of Core-Collapse Supernovae: Long-Term Effects of the Equation of State. *ApJ*, 629:922–932. [4](#)
- Teyssier, R. (2002). Cosmological hydrodynamics with adaptive mesh refinement. *Astronomy and Astrophysics*, 385(1):337–364. [4](#)
- Thompson, T. A., Burrows, A., and Pinto, P. A. (2003). Shock Breakout in Core-Collapse Supernovae and Its Neutrino Signature. *ApJ*, 592:434–456. [4](#)
- Tsatsin, P., Budiardja, R., Cardall, C., Endeve, E., Marronetti, P., and Mezzacappa, A. (2011). GenASiS: A full GR-RMHD simulation framework: overview, goals, and preliminary tests. In *APS Meeting Abstracts*, page 12006. [41](#)
- Wise, J. H. and Abel, T. (2011). ENZO+MORAY: radiation hydrodynamics adaptive mesh refinement simulations with adaptive ray tracing. *MNRAS*, 414:3458–3491. [4](#)
- Wongwathanarat, A., Hammer, N. J., and Müller, E. (2010). An axis-free overset grid in spherical polar coordinates for simulating 3D self-gravitating flows. *A&A*, 514:A48. [4](#), [36](#)
- Woosley, S. and Janka, T. (2005). The physics of core-collapse supernovae. *Nature Physics*, 1(3):147–154. [2](#)
- Zhang, W., Howell, L., Almgren, A., Burrows, A., and Bell, J. (2011). CASTRO: A New Compressible Astrophysical Solver. II. Gray Radiation Hydrodynamics. *ApJS*, 196:20–+. [4](#)



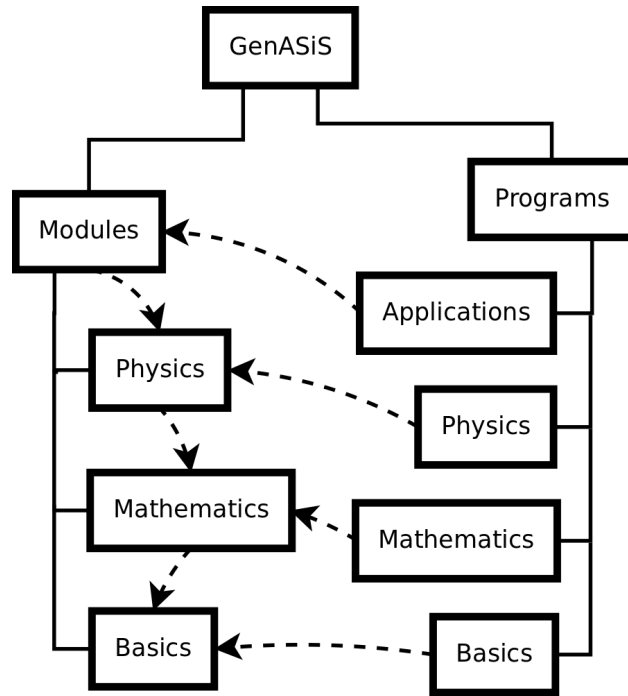


Fig. 1.— High-level structure of the core of GenASiS. Solid lines outline the directory hierarchy, and dashed arrows indicate compilation dependencies.

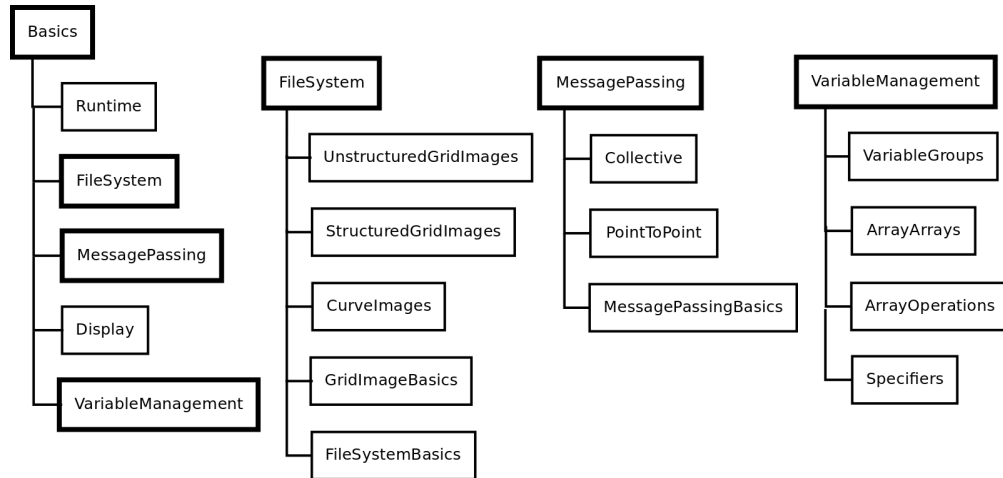


Fig. 2.— *Left:* Structure of Basics. (For the place of Basics in the overall scheme of GenASiS, see Figure 1.) *Middle Left:* Substructure of FileSystem. *Middle Right:* Substructure of MessagePassing. *Right:* Substructure of VariableManagement. *All:* Solid lines outline the directory hierarchy. Boxes framed with thinner linewidths denote ‘leaf’ divisions of the code with no additional subdirectories. The compilation order is from bottom to top; thus dependencies essentially flow in reverse, from top to bottom.

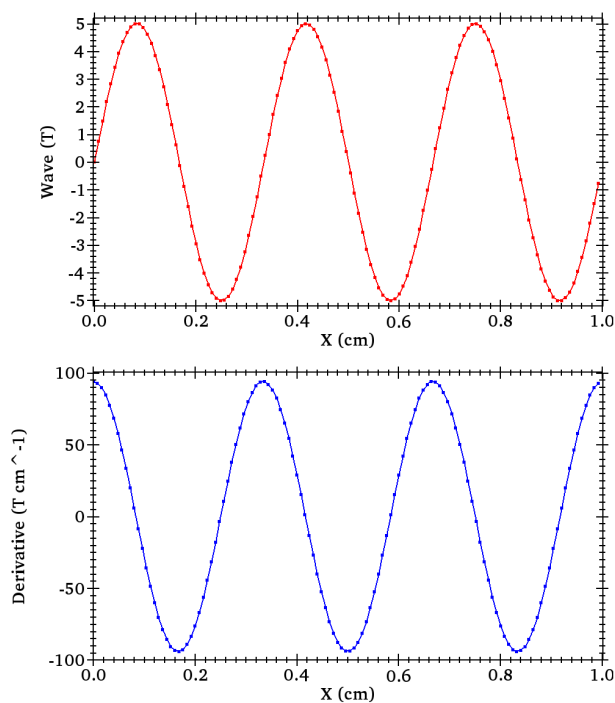


Fig. 3.— Sine wave (top) and its derivative (bottom), with data produced by the SineWaveDerivative example program run with non-default parameters  $\text{Amplitude}=5.0 \sim \text{TESLA}$  and  $\text{nWavelengths}=3$ .

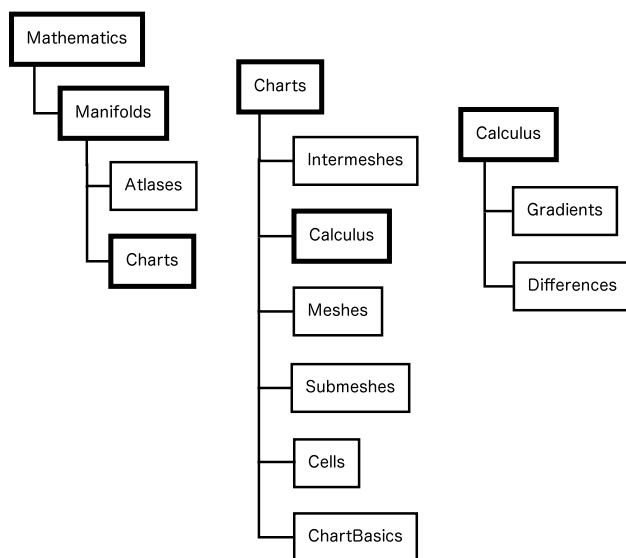


Fig. 4.— *Left:* Manifolds, a division of Mathematics, has divisions Charts and Atlases. (For the place of Mathematics in the overall scheme of GenASiS, see Figure 1.) *Middle:* Substructure of Charts. *Right:* Substructure of Calculus. *All:* Solid lines outline the directory hierarchy. Boxes framed with thinner linewidths denote ‘leaf’ divisions of the code with no additional subdirectories. The compilation order is from bottom to top; thus dependencies essentially flow in reverse, from top to bottom.

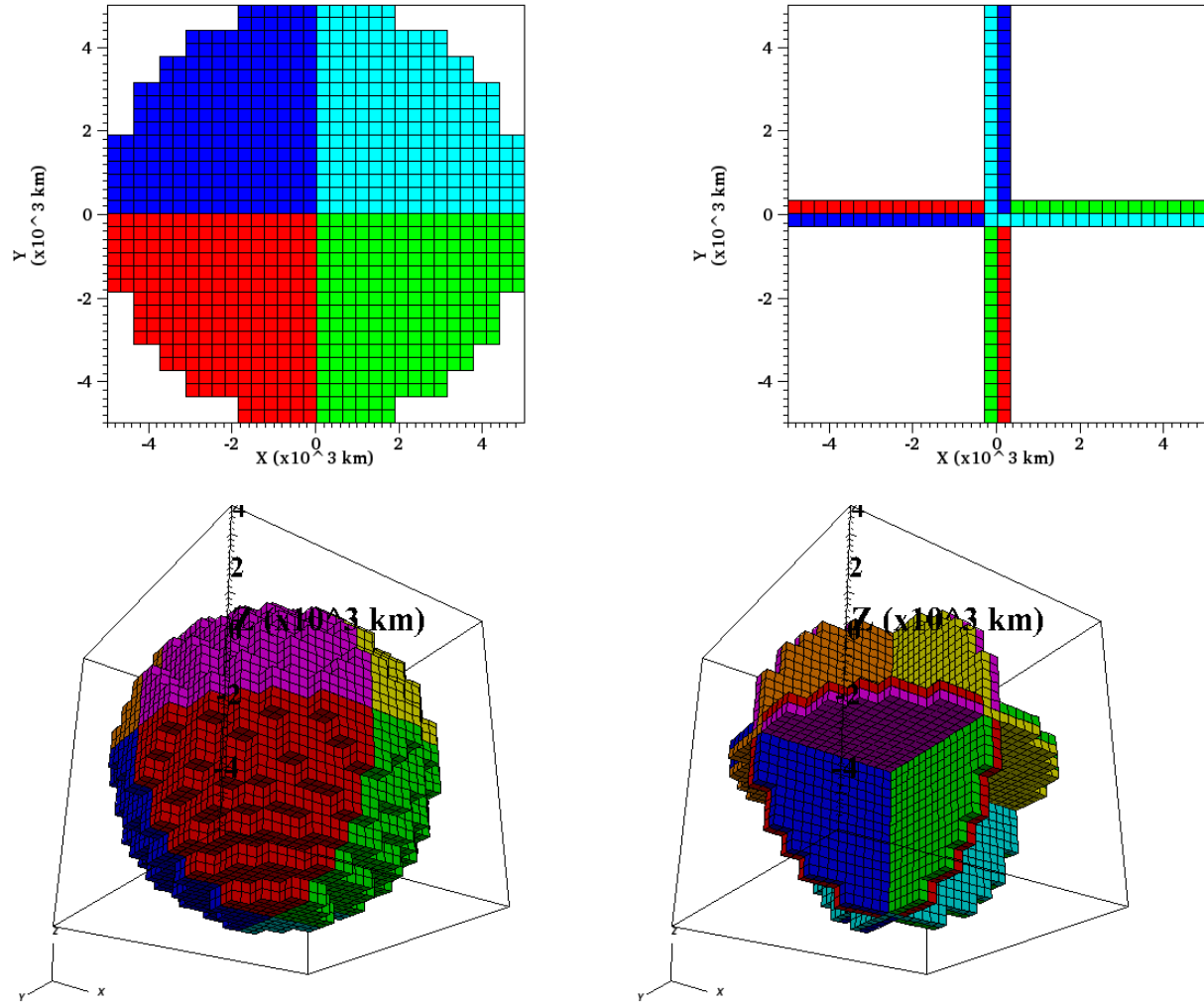


Fig. 5.— Domain decomposition of spherical submeshes in two dimensions (upper panels, four processes) and three dimensions (lower panels, eight processes). Both proper cells (left panels) and the ghost cells partially bounding them (right panels) are shown. Each individual cell is pictured; low resolution and low process counts are used for clarity.

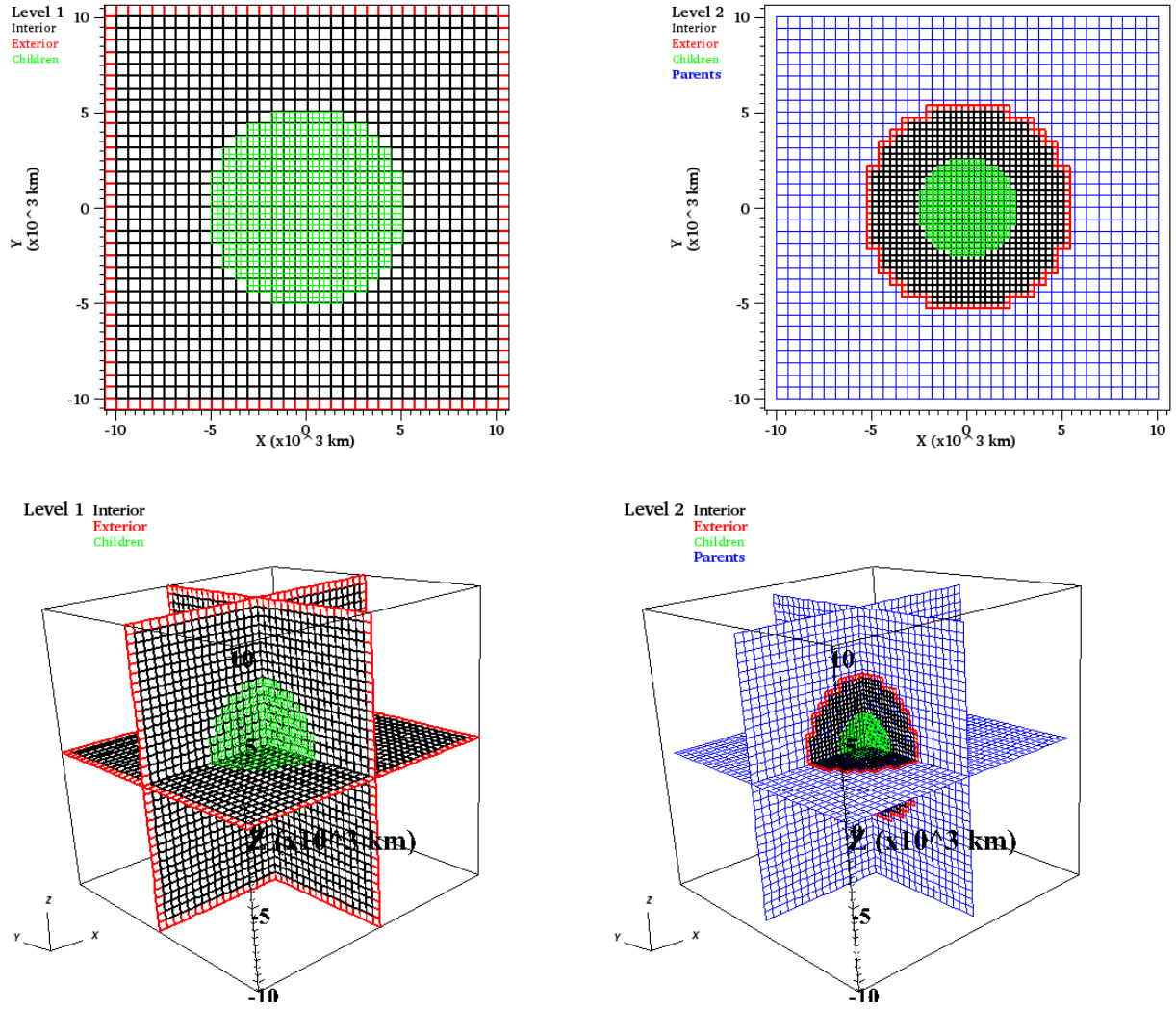


Fig. 6.— Interior (black, thick), Exterior (red, thick), Children (green, thin), and Parents (blue, thin) submeshes, in two dimensions (upper panels) and three dimensions (lower panels, three-sliced), for Level 1 (left panels) and Level 2 (right panels). Being the coarsest mesh, Level 1 does not have a Parents submesh. Each individual cell is pictured; low resolution is used for clarity.

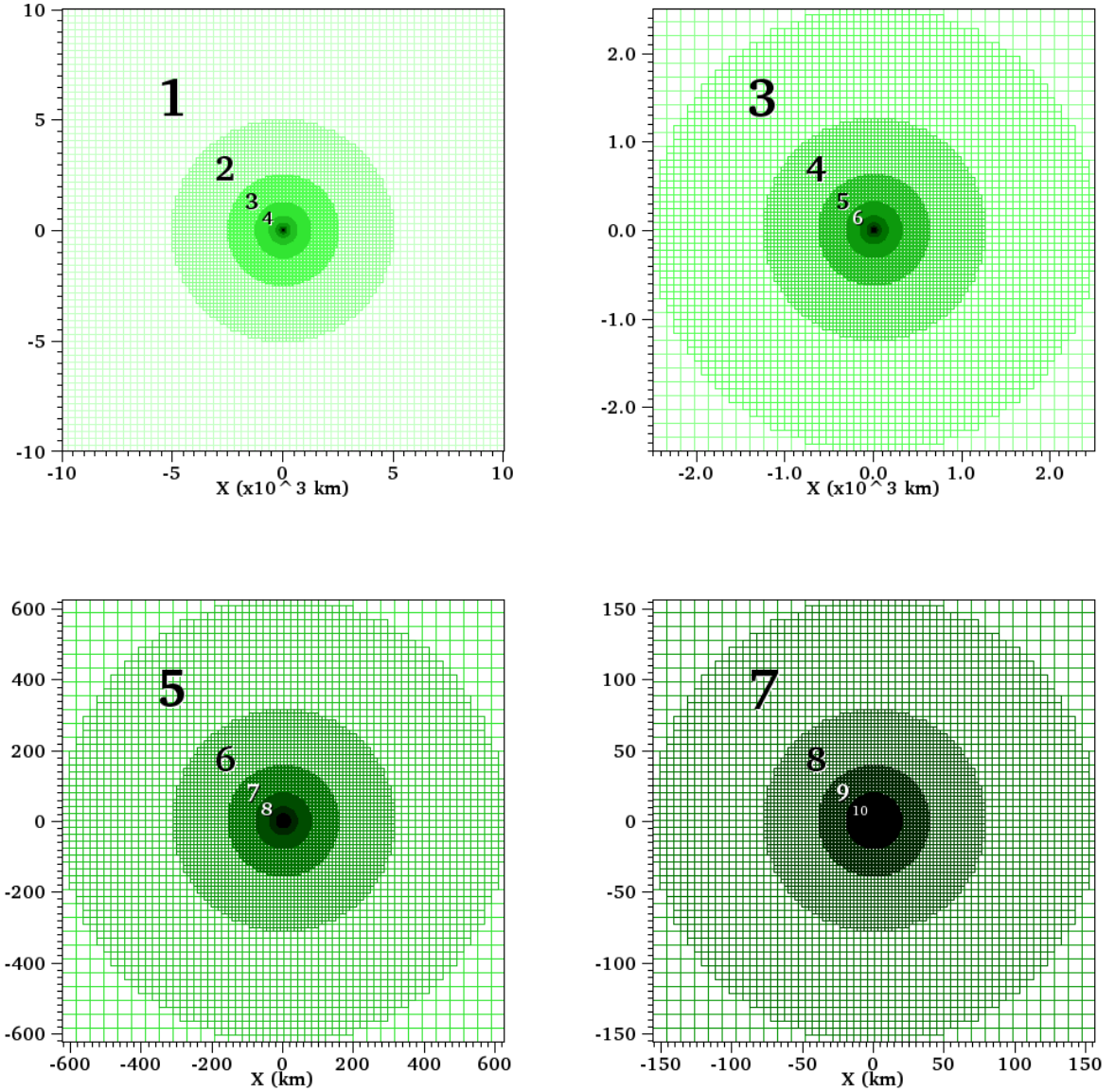


Fig. 7.— Levels of a two-dimensional chart generated by our `GaussianGradient` example program with `nLevels=10`, illustrating the dynamic range in length scales accessed during the gravitational collapse of the core of a massive star. Outlines of  $2 \times 2$  cell arrays, rather than individual cells, are displayed. Level 1—the coarsest mesh—encompasses a square domain of width 20,000 km with  $128 \times 128$  cells of width  $1.56 \times 10^2$  km. Successive levels are refined by a factor of two in each dimension, labeled by their level number, and outlined in increasingly dark shades of green. The upper left, upper right, lower left, and lower right panels zoom in to smaller and smaller regions surrounding the origin. Level 10—the finest mesh—has cells of width 0.305 km.



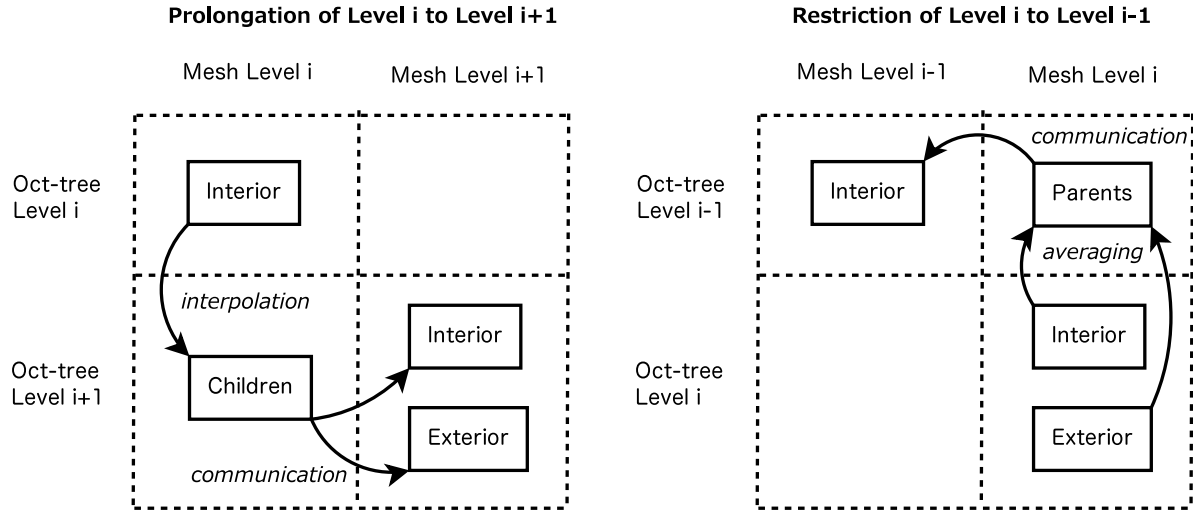


Fig. 8.— *Left:* Prolongation of Level  $i$  to Level  $i+1$ . This involves the Interior and Children submeshes of the Level  $i$  instance of MeshForm (left column), as well as the Interior and/or Exterior submeshes of the Level  $i+1$  instance of MeshForm (right column). The Level  $i$  Interior submesh consists of cells at Level  $i$  of the oct-tree (upper row), while the Level  $i$  Children and Level  $i+1$  Interior and Exterior submeshes consist of cells at Level  $i+1$  of the oct-tree (lower row). Interpolation to the Children is followed by communication to the finer-level Interior and/or Exterior. *Right:* Restriction of Level  $i$  to Level  $i-1$ . This involves the Interior, Exterior, and Parents submeshes of the Level  $i$  instance of MeshForm (right column), as well as the Interior submesh of the Level  $i-1$  instance of MeshForm (left column). The Level  $i$  Interior and Exterior submeshes consist of cells at Level  $i$  of the oct-tree (lower row), while the Level  $i$  Parents and Level  $i-1$  Interior submeshes consist of cells at Level  $i-1$  of the oct-tree (upper row). Averaging to the Parents is followed by communication to the coarser-level Interior.

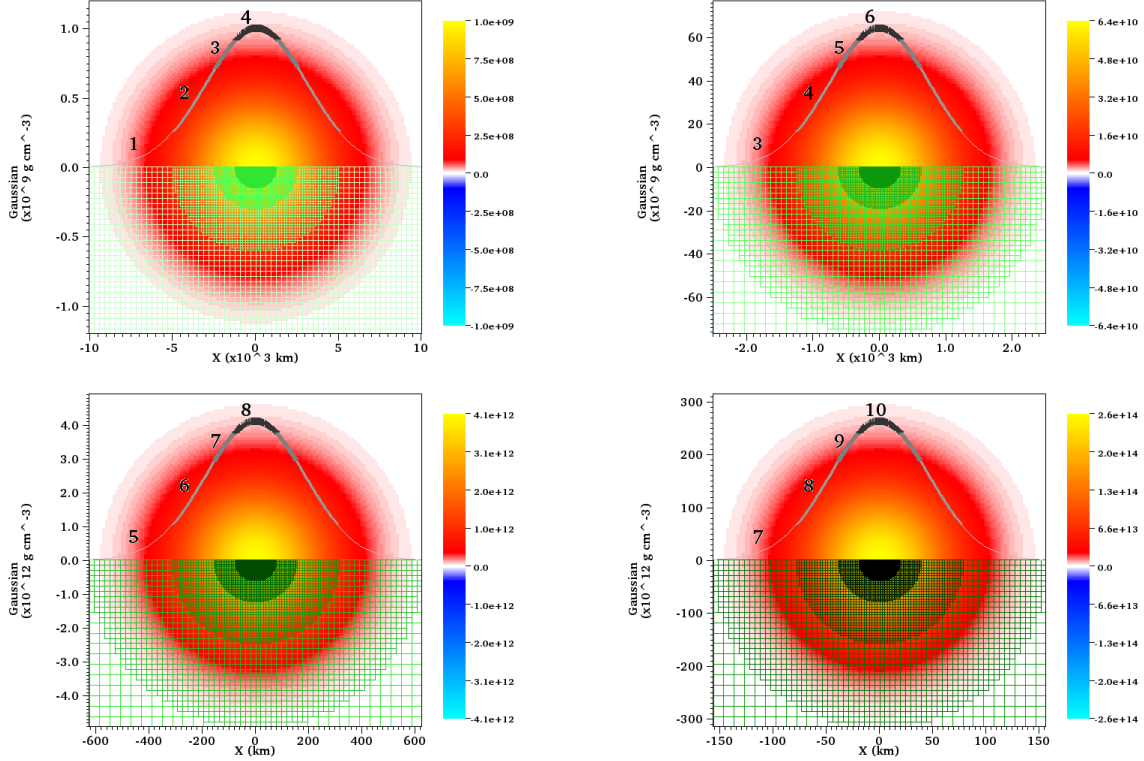


Fig. 9.— Gaussian functions in two dimensions (yellow/red pseudocolor). The upper left, upper right, lower left, and lower right panels zoom in to smaller and smaller regions surrounding the origin. By way of illustrating the range of scales encountered in the gravitational collapse of the core of a massive star, the widths of the Gaussian functions decrease by a factor of 4 (matching the decrease in the figures’ length scales) and the amplitudes increase by a factor 64 from panel to panel. The mesh levels shown in the lower half of each panel are the same as those labeled in the corresponding panels of Figure 7. The curves superimposed in the upper half of each panel show values of the function along the  $x$ -axis, with contributions from meshes of increasing refinement indicated by darker and thicker segments, illustrating the continuity of the function at boundaries between adjacent levels of refinement.

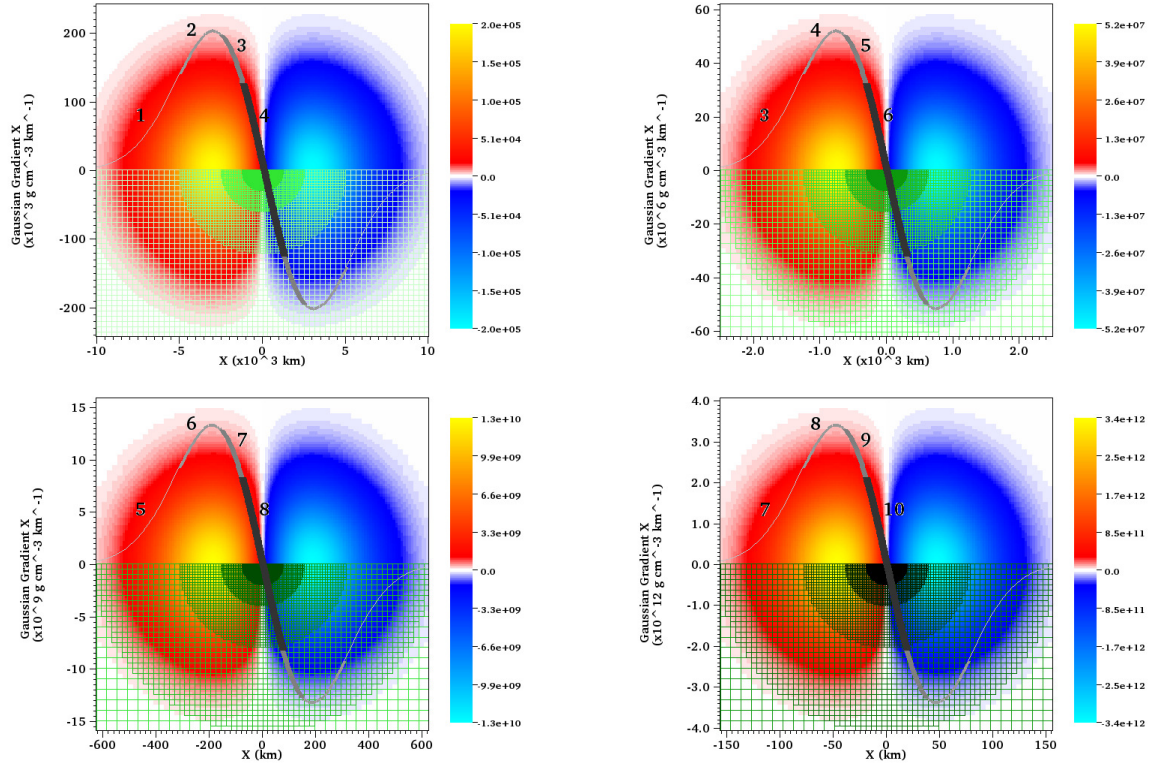


Fig. 10.— The  $x$  component of the gradient of the Gaussian functions depicted in Figure 9 (yellow/turquoise pseudo-color). The demonstrated continuity of the curves along the  $x$ -axis is possible because of the availability of boundary values on Level  $i + 1$  obtained by prolongation from Level  $i$ .

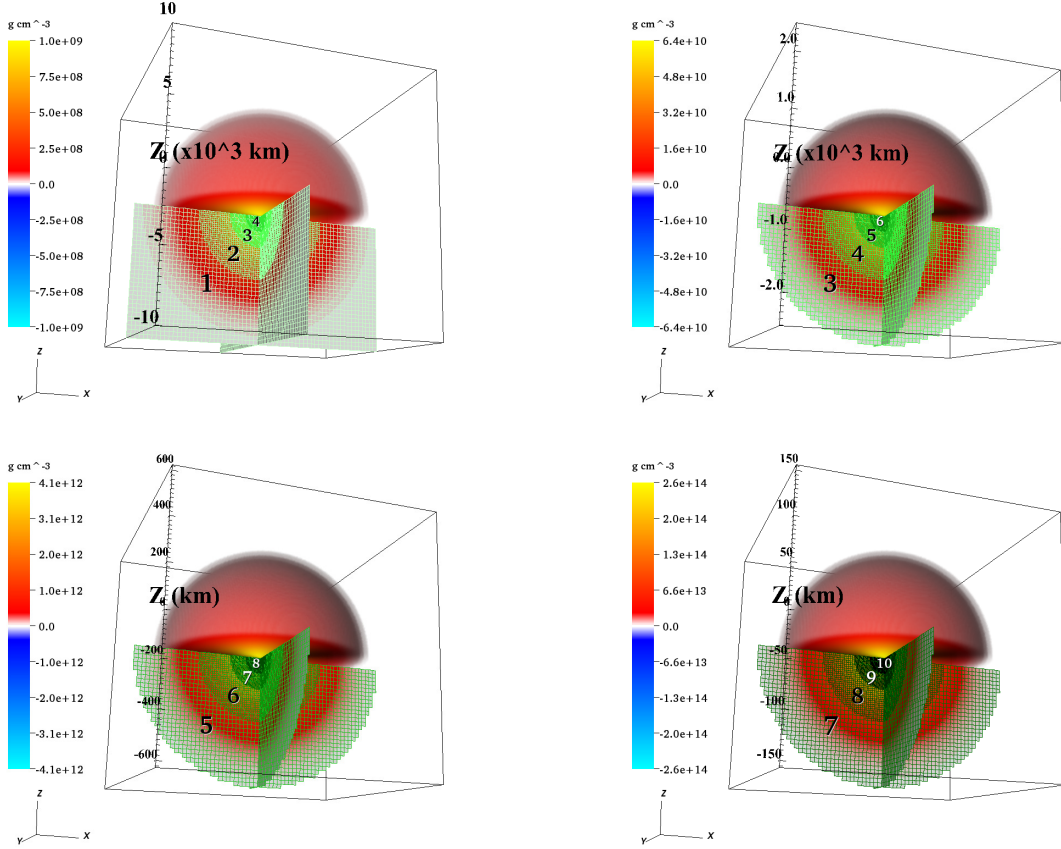


Fig. 11.— Gaussian functions in three dimensions (yellow/red volume plots, clipped to the upper half of each panel). The upper left, upper right, lower left, and lower right panels zoom in to smaller and smaller regions surrounding the origin. By way of illustrating the range of scales encountered in the gravitational collapse of the core of a massive star, the widths of the Gaussian functions decrease by a factor of 4 (matching the decrease in the figures’ length scales) and the amplitudes increase by a factor 64 from panel to panel. The mesh levels shown (three-sliced) in the lower half of each panel are the same as those labeled in the corresponding panels of Figure 7.

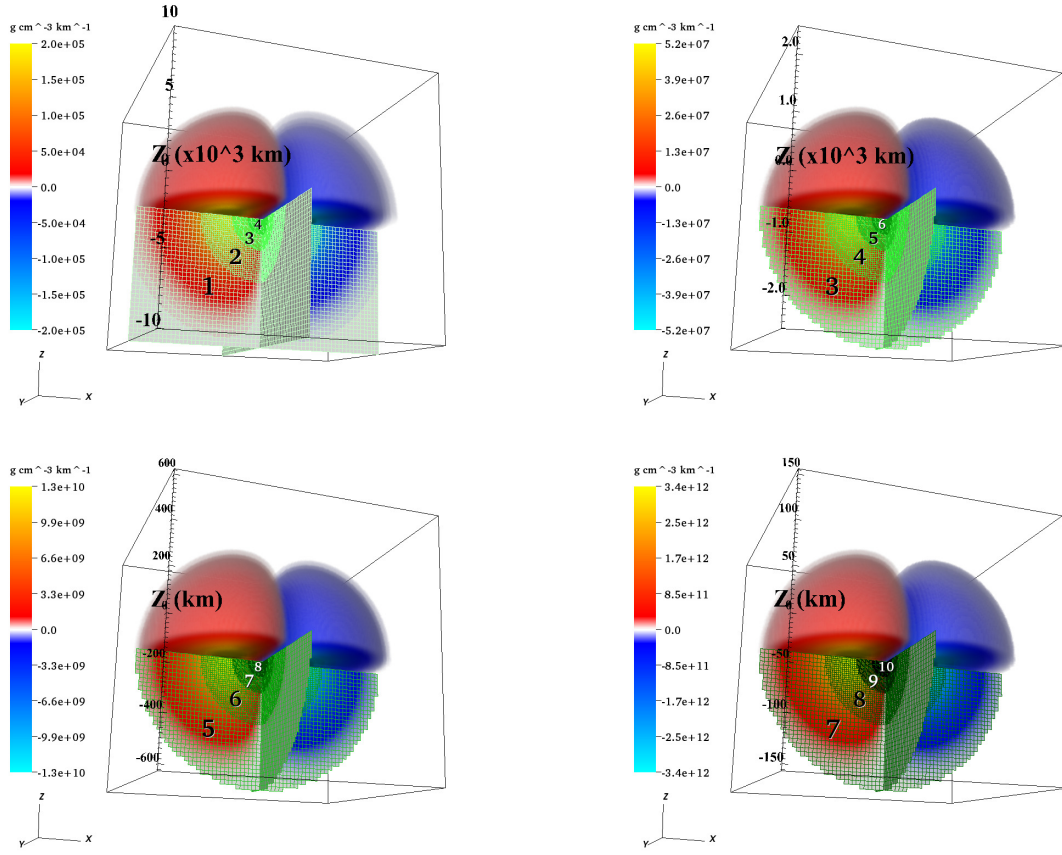


Fig. 12.— The  $x$  component of the gradient of the Gaussian functions depicted in Figure 11 (yellow/turquoise volume plots, clipped to the upper half of each panel).